Visual Modeling with Logo

Exploring with Logo

E. Paul Goldenberg, editor

1. *Exploring Language with Logo* by E. Paul Goldenberg and Wallace Feurzeig
2. *Visual Modeling with Logo* by James Clayson

Visual Modeling with Logo

A Structured Approach to Seeing

James Clayson

The MIT Press
Cambridge, Massachusetts
London, England

This book was printed and bound by Halliday Lithograph in the United States of America.

Library of Congress Cataloging-in-Publication Data

Clayson, James

Visual modeling with Logo.

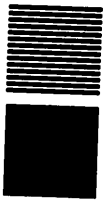
(Exploring with Logo ; 2)

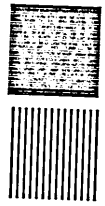
Includes index.

1. LOGO (Computer program language) 2. Computer graphics. I. Title. II. Series.

QA76.73.L63C52 1987 006.6'6 87-3894

ISBN 0-262-53069-4 (pbk.)





To L. A.

Contents

Series Foreword

The aim of this series is to enhance the study of topics in the arts, humanities, mathematics, and sciences with ideas and techniques drawn from the world of artificial intelligence--specifically, the notion that in building a computer model of a construct, one gains tremendous insight into the construct. Each volume in the series represents a penetrating yet playful excursion through a single subject area such as linguistics, visual modeling, music, number theory, or physics, written for a general audience.

Preface

"What is the use of a book," thought Alice, "without pictures . . . ."
Lewis Carroll

In the fall of 1982 I started to teach a course called "Problems in Visual Thinking." It was offered jointly by Parsons School of Design in Paris and the American College in Paris. Looking back now, perhaps I should have replaced the word *Problems* with something less pathological—*Explorations* maybe. But that original title really indicated my reason for inventing the course in the first place. I taught courses in statistics and operations research in which I encouraged my students to add a bit of visual thinking to their quantitative analysis, but each semester I was disappointed.

I continued my pleas for visualization because I saw that the few students who could introduce a little of it into their work discovered—more often than not—the most surprisingly useful things. These visualizers seemed to me less intimidated by vagueness because their picture-making abilities gave them concrete starting points, and they seemed to enjoy playing around with the painted pieces of complex problems. Perhaps, I thought, their visual play encouraged them to *see* where more analytic approaches might usefully be applied.

My problem was to discover how to teach visual thinking to those students who had problems doing it naturally. It was obvious that most of my students lacked visual vocabulary and few of them had ever been in an art or drawing studio. How was I to cancel out this liability? Luckily I had managed an art

school and knew a bit about people who had design experience. Professional art students certainly have the visual baggage, but most are severely lacking in analytical skills. "Let's put these two groups together, " I thought, "and set them a series of tasks." The art students can show their colleagues a bit about color and design, while the non-art crew can gently introduce the art students to a little quantitative model building. The Logo computer language struck me as an appropriate medium of instruction—just enough of the visual and just enough of the analytical. Problems in Visual Thinking was born.

There were no suitable texts, so I set out to write one, and this book is the most recent set of class notes. It is structured around a series of exercises that encourage visual thinking in students from a variety of different backgrounds.

I wish that I could claim total success in turning my students into better problem-solvers by first turning them into more effective visualizers. But I fear that my record is mixed. I am convinced, however, that for some people, certainly not all, visual model building is an enormously enjoyable activity that leads them in new and surprising directions. And since that activity falls nicely within the terms of reference of a liberal arts education, I am quite pleased with the classroom results I have seen.

Most thanks are due to my students because this book was realized with their help. You will find quotes and illustrations from them scattered throughout the text. Thanks, too, go to Roger Shepherd, the first Director of Parsons in Paris, who not only encouraged me to start this project but helped to teach it for the first year. Were it not for Frank Satlow of MIT Press, this book would still be in a basement Xerox room. I also benefited from his readers' reports.

The final construction of the manuscript, however, was a solo affair; whatever opacities, inconsistencies, or mistakes remain are mine.

James Clayson
Paris, 1987

Acknowledgments

I wish to thank the following for granting permission to reproduce illustrations in this book:

Robert Delaunay, *Disque,* Musée d'art moderne de la ville de Paris, courtesy of ADAGP and ARS.

David Hockney, *Sunbather,* courtesy of the artist.

Wassily Kandinsky, *Several Circles, No. 323* (1926), Solomon R. Guggenheim Museum, New York. Photograph by Robert E. Mates.

Wassily Kandinsky, *First study for Several Circles* (1926), Collections du Musée National d'art moderne, Paris, courtesy of ADAGP and ARS.

Gustav Klimt, *Poissons rouges,* courtesy Giraudon/Art Resources, New York.

Gustav Klimt, *Rosiers sous les arbres,* courtesy Giraudon/Art Resource, New York.

Piet Mondrian, *Compositie met kleurvlakjes no. 3,* Collection Haags Gemeentemuseum, the Hague, courtesy of Beeldrecht, The Netherlands/VAGA, New York.

Piet Mondrian, *Composition with red yellow and blue,* Tate Gallery, London, courtesy of SPADEM and ARS.

Piet Mondrian, *New York City 1*, Collections du Musée National d'art moderne, Paris, courtesy of Beeldrecht, The Netherlands/VAGA, New York.

Georges Seurat, *La Seine à la Grande-Jatte*, Musées royaux des Beaux-Arts, Brussels.

Jean Tinguely, *Homage à Marcel Duchamp*, courtesy of the Städtisches Museum Abteiberg Mönchengladbach, photo by Ruth Kaiser.

Henry van de Velde, *Faits du village VII–La ravaudeuse*, Musées royaux des Beaux-Arts, Brussels.

Vincent van Gogh, *Van Gogh's bedroom*, courtesy Giraudon/Art Resources, New York.

Andy Warhol, *Marilyn Monroe*, courtesy Giraudon/Art Resource, New York.

Architectural trees reproduced from Bob Greenstreet, *Graphics Sourcebook* (Prentice-Hall, Englewood Cliffs, NJ, 1984).

Celtic knots reproduced from George Bain, *Celtic art: the methods of construction* (Dover, New York, 1973).

Computer-generated globes reproduced from Melvin L. Prueitt, *Computer Graphics* (Dover, New York, 1975).

Farkas grids reproduced from Tamás F. Farkas and Péter Erdi, "Impossible forms: experimental graphics and theoretical associations," *Leonardo*, volume 18, number 3 (1985), pages 179-183, copyright © 1985 ISAST.

Islamic tile designs reproduced from J. Bourgoin, *Arabic geometrical pattern and design* (Dover, New York, 1973), and from Keith Critchlow, *Islamic patterns* (Schocken Books, New York, 1976).

Lattice designs reproduced from Daniel Sheets Dye, *Chinese lattice designs* (Dover, New York, 1974).

Stone mason marks reproduced from Matila Ghyka, *The geometry of art and life* (Dover, New York, 1977), and Charles Bouleau, *Charpentes, La Géométrie secrète des peintres* (Editions du Seuil, Paris, 1963).

Tree silhouettes reproduced from Derrick Boatman, *Fields and Lowlands* (Hodder & Stoughton, London, 1979), courtesy of the Rainbird Publishing Group.

Visual Modeling with Logo

Chapter 1
Introduction

"Skill to do comes of doing."
Ralph Waldo Emerson

"Chance favors the prepared mind."
Louis Pasteur

Who I hope you are

You might be interested in the studio arts—painting, drawing, sculpture, photography—or in architecture. You may have had experience in graphic, commercial, or industrial design. Then again, you might be a liberal arts type with a background in language, literature, music, or science. Or perhaps you are a professional type educated in the field of business, law, medicine, or theology. You could be a student or a teacher or neither or both.

You may have spent long hours in art studios and possess an exceptionally rich, visual vocabulary. Then again, your graphic abilities, both verbal and physical, may be very much on the thin side. Instead, your vocabulary might be skewed toward logic and mathematical terms because your background is in the sciences, philosophy, law, or mathematics. Perhaps you are that much sought after, well-rounded person whose vocabulary is rich without being specialized.

You may not be able to describe every phase that Picasso went through, but you enjoy looking at art, and you can differentiate a Picasso from, say, a Pissarro when you see them side by side. You probably have a favorite artist or a favorite period, and you have paintings or reproductions of them in your own home. While you may not be able to sketch the Acropolis using 3-point perspective, you do have some idea of what perspective means. You would be intrigued by the suggestion that, in fact, there are dozens of different ways to illustrate objects in space.

At some time in your past, you must have taken a course in geometry. (Everybody has taken a course in geometry. It is one of the few bits of liberal education that we still share.) What about trigonometry? You may have forgotten everything, but you aren't brought to the edge of coronary arrest by hearing the words *geometry* and *trigonometry.*

Finally, and most important, whatever your curriculum vitae says, and whatever type of intellectual or visual baggage you carry, *I hope that you are excited by looking at things and by thinking about how things look.*

This book will show you how to increase this kind of excitement by encouraging you to build a special, visual variety of computer models. If you are stimulated by this idea, then you are exactly who I hope you are.

Your computer baggage

This chapter gives a very quick summary of basic Logo. It is brief, not just because I hope that you already know a little Logo but because I expect that you are willing to use the manual that came with your copy of Logo. In other words, I expect that you are willing to do some learning about Logo mechanics on your own.

If you have never met Logo before, probably you have been introduced to some other computer language and most likely this other language was BASIC. (I am saddened to think of you learning BASIC before Logo, and this book will

give you the chance to right that terrible wrong.) If you do know something about one computer language, you should be able to plunge happily into the manual of a second language—Logo. I will give you some directional help, though, by suggesting what questions you need answered in your manual. Then it will be up to you to learn the specifics.

I assume, too, that you have played around with personal computers and know how they "feel." You may like or dislike these machines, but your feelings are based on personal experience. You are familiar with disks, disk drives, keyboards, and program editors.

No baggage

If you have had no experience with personal computers, and have never tried to learn a computer language—on your own or in a course—you may find this book rough going. If, on the other hand, you have access to a teacher, tutor, or friend who is willing to give you help when you ask for it—and if you are patient, curious, and tenacious—I think you should stick around.

Programming as craft

Learning to program in Logo is very much like learning a craft. You can read about "doing Logo" as you can read about making furniture, and you can talk to others about doing it as well. But in order to develop the individual talents and skills needed for effective Logo or furniture craftsmanship, you must physically do it yourself.

For the newcomer to a craft, a master artisan can certainly be of help. The old timer can suggest small, beginning projects that are reasonably taxing but not overly intimidating. And by offering encouragement, he can keep the novice's spirits high.

This chapter is designed to reinitiate you into the excitement of Logo craft; all the exercises within the chapter have been fashioned by an experienced craftsman. These initial projects should be copied. All artisans begin learning their craft by copying what others have done. This is not to deny their creativity but rather to allow for a strengthening of the basic skills that support individual creativity. Good craftsmanship, of course, requires both skill and creative flair.

For my purposes, skills are as much frames of mind as forms of physical dexterity. For example, in this chapter I will stress the usefulness of three such skills: first, an ability to break down big problems into smaller problems; second, a willingness to imagine yourself as "walking" shapes into existence; and third, a propensity to tinker with your Logo machinery. Very soon (starting with the exercises at the end of this chapter, in fact) you will be asked to apply these skills to support your own kind of invention.

Craft is about building things by hand, and that is what you will be doing with Logo.

Copying and tinkering

Copying computer programs is so widespread that I had better give you my opinion about the usefulness of it all. In the previous paragraph I said that copying was a good thing at the start of one's apprenticeship. As you go through this chapter, I hope you will want to try out my procedures. That's OK. But if you simply type my procedures into your computer, you will only reproduce what I have done. And that will bore us both. The book shows what I have done; it doesn't show what you can do. My procedures offer you a starting place, so that you don't have to build from zero. But it is up to you to go beyond that start.

So please do copy the *ideas* of any procedure that strike your fancy. But, once you have made those ideas work on the screen, play rough with them. Give the copied procedure funny and outlandish arguments. (Very big or very small

numbers could be outlandish, but what is a funny argument?) What happens then? Does your copied procedure still work? Can you explain how? Go on to make a few changes inside the body of the procedure; change some of the commands just a little. Can you guess what might happen before you experiment on these changes?

Get into the habit of *tinkering*, just a little. Whenever you write, or copy, a nice procedure, make a few changes to it so that it does something else.

Equipment

Your most important piece of equipment is a notebook. It is far more important than the computer you work on and all the technical manuals at your disposal. Select a notebook with large, unlined, and bound-in pages. I want you to keep track of your work in this notebook. That means everything: the little sketches, word portraits, diagrams, procedure listings, the printed images. Stick in any magazine and newspaper illustrations that strike your fancy, whenever you find them; don't worry about organization. You will be mixing the good with the less than good, the things that worked well with those that never will.

I suggest that a large format notebook is best. That means plenty of room for a lot of stuff. Small notebooks encourage crabbed handwriting and get messy; you will need a lot of space. Also, unlined paper is better. Since there is no need to write carefully, lines will get in the way. If you need graph paper for a careful diagram, glue a piece in. And, finally, the bound-in pages will not let you reorganize the book. The notebook may organize you.

Glue. Rubber cement is the Queen of Glues and the world's best notebook adhesive. Get a lot of it. Be careful using this glue, though; it is very flammable. Don't smoke and glue at the same time.

Dialects

Unfortunately, there is not just one Logo. While some Logos are more alike than others, most have quirks.[1] I use Terrapin MacLogo throughout this book. All the procedures have been written in this dialect using an Apple Macintosh Plus. Most of the images were generated by Logo procedures and printed on an Apple Imagewriter II printer. The rest were done by hand, mine.

You may have a different machine and a different Logo. To make life as easy as possible, and to eliminate the need to talk about dialects, I have tried hard to avoid using those components that vary most between Logos. The bad news: this is a book about graphics and graphics is the area in which Logos differ most. So I skirt any graphics razzle-dazzle that might work in one Logo and not in another. I don't use funny pen patterns, polygons filled with patterns that look like brick walls or tile roofs, or automatic mirrors that reverse images. There are no sprites or multiple turtles. The only colors I use are white, black, and (very occasionally) a reversing pen color. All the line drawings have been done with a standard, narrow-width pen.

Caveat emptor

This is a little late for a warning, but here it is anyway: *This is not a book about Logo.* You will end up knowing a lot about Logo, and that is no bad thing. But the goal of the book is to get you to build visual models, and Logo is only a means to that end. God knows, we could have used Pascal. But it just so happens that Logo is easier to learn and easier to use than most of the other languages that we

1. Appendix A in Brian Harvey's *Computer Science Logo Style*, volume 1: *Intermediate Programming* (Cambridge, MA: MIT Press, 1985), gives a nice summary of the syntactic differences between Logos. It gives no help with differences in graphics, though.

could have selected. Logo notation is neat and tidy; it looks nice on the page and that encourages visual thinking. But most important, because Logo is so easy to play around with, it won't get in the way.

Turtles are us

We will concentrate on the graphic parts of Logo. And that means "turtle graphics." The intent of Logo's turtle metaphor was to inspire young children to explore shapes. The turtle is a tiny triangle of light that is moved about the screen via Logo commands. As the turtle moves, it leaves a trace of light. Children are encouraged to imagine themselves in the turtle's place and to draw a shape by walking through it, as the turtle would walk through it.

Children have the necessary body knowledge to walk a circle, even though they cannot express their circle drawing rules before walking them. Walking the turtle around an invisible circle translates the body's knowledge into word commands: "I'm walking him forward a little bit, now I'll turn him a little, I'm walking him forward a little bit, now I'll turn him a little . . . I'll keep doing this until until he's finished. Yes, that's it; I'm back where I started." Once said, the words are available to be transformed into Logo commands.

Because adult bodies may be more spatially intuitive than children's, your turtle visualizations can be far more effective than a child's. There is a problem with adults, though; they aren't used to playing imagination games as adults and must be coaxed into it. Children are happy to play silly games; adults may be embarrassed to try. I will be asking you, after all, to imagine yourself as an electronic turtle. And what would that feel like, I mean physically? Use your turtle body and walk around a bit.

Don't reject visualization and muscular thinking before you try it. What was good enough for Uncle Albert should be good enough for you. Listen to what he said: "The physical entities which seem to serve as elements in thought are certain signs and more or less clear images which can be 'voluntarily' reproduced

and combined . . . this combinatory play seems to be the essential feature of productive thought. The above elements are, in my case, of the visual and some of the muscular type."—Albert Einstein

Turtle space

Turtles live on your computer screen. Make sure you know the size of yours, since different computer screens have different dimensions. Screen dimensions are generally stated in vertical (the y-axis direction) and horizontal (the x-axis direction) measurements. Pinpoint the x = 0 and y = 0 point on your screen.

The turtle can be moved about the screen using cartesian x-y coordinates or turtle coordinates. Cartesian commands send the turtle to a specific xy position on the screen, without regard to the turtle's current position.

Various `SET` commands move the turtle through cartesian space. Review them.

In the turtle reference system, all commands refer to the turtle's current position, not its final position. The turtle is moved forward, backward, turned left or right in relation to where it is now.

Review the turtle reference commands: `FD, BK, PU, PD, RT LT.`

In the cartesian system, the destination is the important thing; in the turtle reference system, it's the trip.

Making shapes

Let's draw a simple shape using turtle reference commands. Suppose you would like the turtle to draw a square box located at the center of the screen (usually the origin of the xy coordinate system). Here are the three steps you would take:

First, you would clear the screen by typing `CG` (clear graphics). The turtle now sits at 0,0 and faces straight up.

Second, you would think about the commands needed to walk the turtle through the shape. *Use* the turtle metaphor.

1. "OK turtle? Go forward 50 steps and turn right by 90 degrees. That completes the left side of the box."

2. "Now, go forward another 50 steps and turn right by 90 degrees. That completes the top of the box."

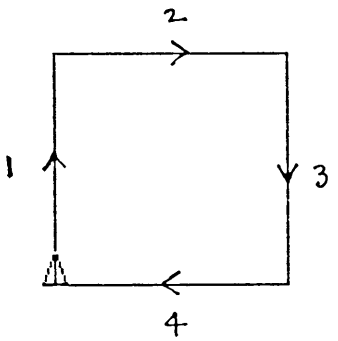
3. "Go forward yet another 50 steps and turn right, again by 90 degrees. That completes the right side of the box."

4. "Go forward another 50 units and turn right 90 degrees. That completes the bottom edge of the box."

That's it. The turtle has walked around the four sides of a size 50 box, arriving back to where it started. These prose commands would translate into Logo as:

```
1. FD 50 RT 90
2. FD 50 RT 90
3. FD 50 RT 90
4. FD 50 RT 90
```

The third and last step would be to type these commands on the keyboard. And here is what you will see.



Great. This series of commands does indeed draw the square you wanted; but wasn't it tedious to type in all that stuff? The command `FD 50 RT 90` was typed four times. Surely there is a shorthand method to repeat this line four times without typing four times. Review the `REPEAT` command; it is exactly what we need here. Try it.

```
CG
REPEAT 4 [FD 50 RT 90]
```

Notice that the line `REPEAT 4 [FD 50 RT 90]` is a kind of operational definition of what a square is: a square is four sides, four `FD` commands, with each side joined at right angles to the next, the `RT 90` commands. That's tidy, but it's still a bore to type two lines each time you want a size 50 box on the screen. After all, you may want to draw 100 boxes.

Wouldn't it be convenient to be able to "tell" Logo your definition of a square and then to give that definition a name?

Logo procedures group commands under a single name

You can group a series of Logo commands together under a single name by writing a Logo procedure. The name of the procedure is a shorthand for all commands in-

cluded in it. Typing the name of the procedure tells Logo to automatically execute each line of the procedure in turn, just as if you had typed them, one after another, on the keyboard.

You can "tell" Logo your definition of a square by creating a procedure called SQUARE. Logo will "remember" your definition until you either erase it or turn off your computer.

Review the defining and editing procedures in your Logo manual.

Shapes defined and drawn by procedures

Let's get on with writing the necessary procedure. Here it is:

```
TO BOX50
 REPEAT 4 [FD 50 RT 90]
END
```

Logo will add BOX50 to all its other commands. Each time you type BOX50, the turtle will draw a square of size 50. The figure will be drawn at the turtle's current position on the screen. Unless you move the turtle to a different starting point, each time you type BOX50, the square produced will be on top of the previously drawn figure. So move the turtle around to new positions and draw some more boxes.

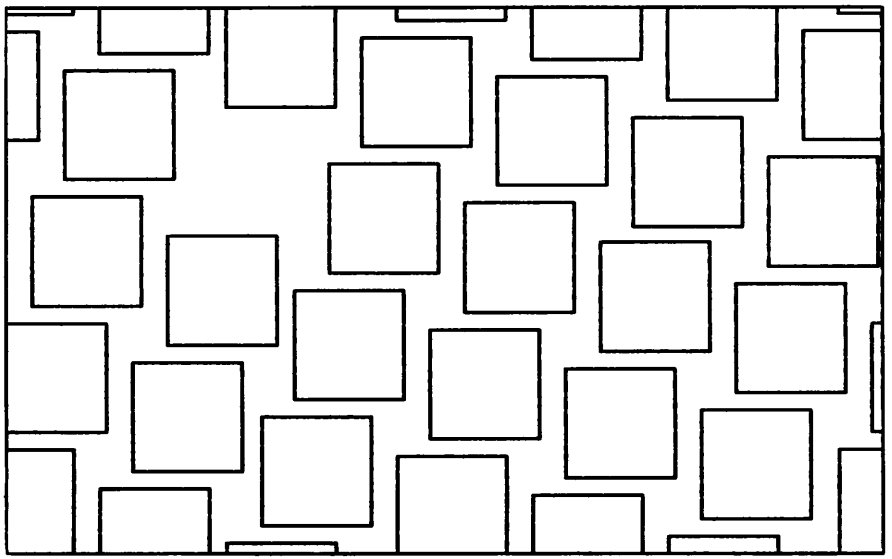
But you can certainly be more imaginative than that. Create an interesting design on the screen using only the BOX50 procedure and move commands. If you have a color screen, you might want to investigate the effects of changing the screen's background color and the color of the pen. Keep track of what you are doing in your notebook so that you can reconstruct your successful designs.

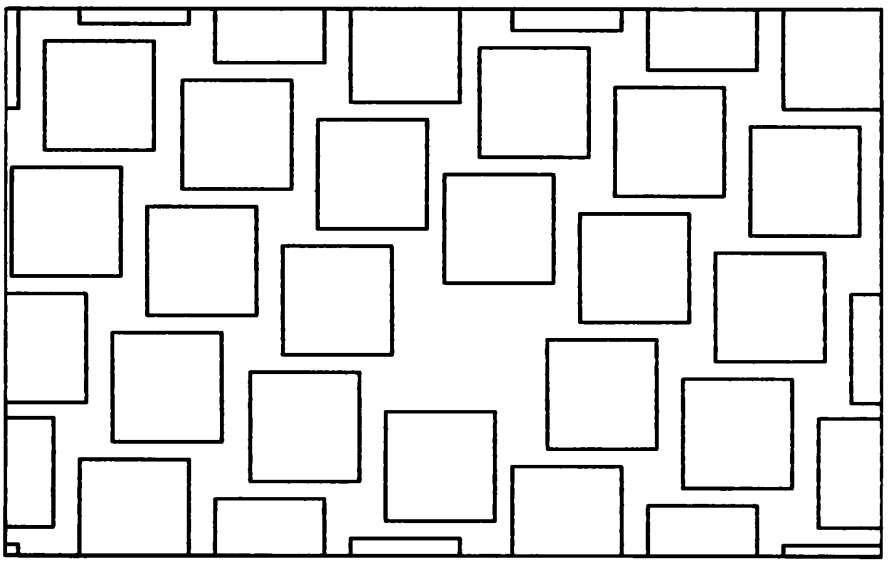
Here is a simple command that wraps boxes around the screen:

```
REPEAT 25 [PD BOX50 PU RT 15 FD 60 LT 15]
```

Wrapped boxes

Here are two different images produced by that one-liner. How are they different? Is one image more pleasing than the other? Why?





Generalizing procedures

What about boxes of different sizes? You could edit the BOX50 procedure every time you wanted it to draw a different size box. You could also define many BOX-like procedures, each to draw a different size box. But that doesn't seem very efficient, does it? After all, Logo itself doesn't have a different FD command for every possible length of line that you might wish the turtle to draw.

There is not a FD-10 command for drawing lines of length 10 and a FD-43 command for drawing them 43 units long. Logo has a single FD command. Whenever FD is used, an argument must be used in conjunction with it: the form is FD argument. A single argument must be typed just after FD. For example, one could type: FD 10 or FD 43. The value of the argument "tells" FD how to go about its business of drawing straight lines. Isn't this convenient? One command does a variety of things. FD argument draws straight lines of *any* length. If we change the value of the argument, the line length changes accordingly.

Let's *generalize* BOX50 in terms of box size as FD is *general* in terms of line length.

Adding an argument to a procedure

Define a new box procedure that has a size argument. The value of the argument will tell the box procedure how to go about its business of drawing boxes. Changing the value of the argument will change the size of the box. You can now draw boxes of any size from, say, 1 unit to 5000 units. Use this example as a "pattern" for incorporating an argument into a procedure.

```
TO BOX :EDGE
 REPEAT 4 [FD :EDGE RT 90]
END
```

Putting a demonstration procedure together

I am sure that by this time you have already designed some interesting patterns with various BOX procedures. Some of these patterns you probably liked enough to print and glue into your notebook. Remember to include a few written comments on what you were trying to achieve.

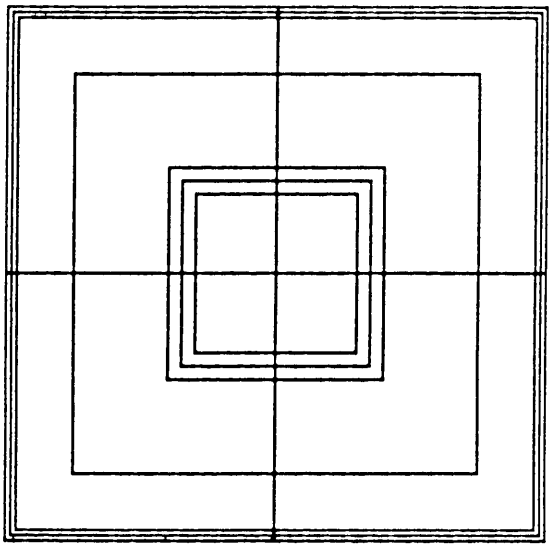
Perhaps you would like to show off your designs. You could show your pals the images in your notebook. But seeing an image is not the same as seeing how the image is drawn, the order in which the pieces are visually assembled. How would you go about demonstrating this? You could retype all the Logo commands needed for your screen collage. What else could you do?

Remember that Logo procedures can group a series of commands together under one name. So let's define a new Logo procedure that will run all the necessary steps to demonstrate your design. Once defined, you will only have to type the demonstration procedure's name to have your designs redrawn on the screen. Your demonstration procedure will have no arguments; it will only do one thing: generate a specific design that you want to show to your friends. If you wish to show off with several designs, you could design specific Logo procedures to reproduce each design.

Here is an example of such a demonstration procedure:

```
TO DEMO
 ; A demonstration procedure to show off a design
 ; produced from multiple boxes of different sizes.
 REPEAT 4 [BOX 100 BOX 98 BOX 96 BOX 75 BOX 40
           BOX 35 BOX 30 RT 90]
END
```

To see the pattern defined by DEMO, type it:



Funny feelings

Do you have a funny feeling that this isn't enough, that we aren't producing great enough images? Let's think about this for a while.

Drawing with Logo should not be the same as drawing with a pen or pencil. What can be sketched quickly by hand is unbearably tedious to sketch with Logo. You may have tried sketching with Logo by moving the turtle as you would move your sketching hand. It doesn't satisfy, and it doesn't work. Logo is a unique medium for visual expression; don't expect it to be like other media. Visual modeling with Logo is as different from drawing as clay modeling is different from photography. Our Logo media is a visual modeling media. We use it best to build models. Why? To encourage us to think about shapes. The drawing is done to encourage thinking.

So don't worry too much (now) about the final image. Don't worry if your designs aren't amazingly beautiful; don't be concerned if they aren't "arty." This isn't, after all, a book about computer "art," but it might be useful to think about ART for a minute. Here is a quote from a computer art type, Harold Cohen from the University of California at San Diego: "For most people outside of art,

probably, art is directed primarily at the production of beautiful objects and interesting images; and who is to argue that a complicated Lissajou figure is less beautiful than an Elsworth Kelly painting or a Jackson Pollock; or that a machine simulation of Mondrian is less interesting than the original it plagiarizes? To talk of beauty or of interest is to talk of taste, and matters of taste cannot be argued with much profit. The fact is that art is not, and never has been, concerned primarily with the making of beautiful or interesting patterns. The real power, the real magic, which remains still in the hands of the elite, rests not in the making of images, but in the conjuring of meaning."

A little professorial, this. But do you think he has a point?

Go back to your demonstration procedures, the ones that have *no* arguments so they do only *one* thing. A procedure that does only one thing is like a box drawing procedure that draws only one size of box. One box doesn't encourage much thinking about the nature of boxes, does it? Can you use your demonstration procedures to explore the nature of a collage that intrigues you? Play around with the collage demonstration procedure that draws it. Tinker a bit.

Maybe you could generalize the demonstration procedure by adding an argument. To generalize a procedure is to stretch your thinking about what it does; and that's our appropriate work, too, because it respects the uniqueness of the Logo art medium. "It's the tinkering that counts, not the artiness." Pin that phrase over your computer screen.

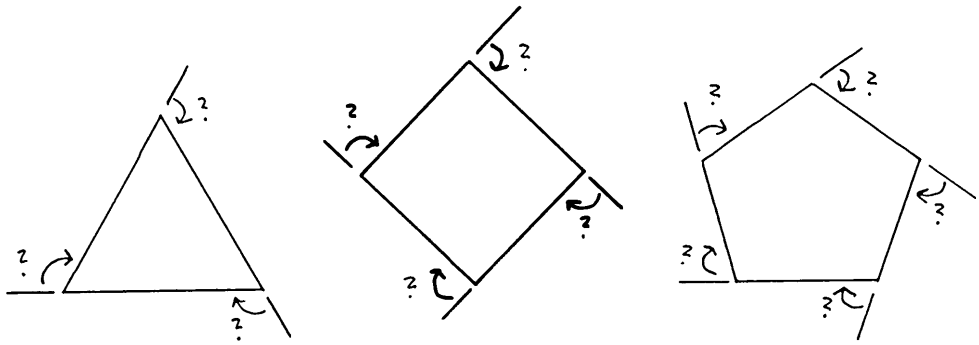
Generalizing a procedure with arguments

Let's go back to that BOX procedure. Can we generalize it so that it draws triangle "boxes" as well as square ones? While we are at it, let's ask BOX to draw boxes with any number of equal sides. These shapes will be regular polygons with n sides. Why not call the generalized procedure NGON for n-sided polygon? Look again at the procedure BOX and decide what needs to be changed to turn BOX into NGON.

```
TO BOX :EDGE
 REPEAT 4 [FD :EDGE RT 90]
END
```

You need to add a second argument :N. This will tell Logo how many sides to draw. We can replace the REPEAT 4 with REPEAT :N. FD :EDGE will stay the same, but what about the angle you want the turtle to turn before drawing the next side? It surely will be different for different sided polygons.

Generally, sketching focuses geometric thinking. Here are some walking plans that you might issue to the turtle to do NGONs.



How can you calculate the angle indicated by the "?" for any n-sided polygon? Your geometric intuition should tell you that the turtle, after making :N turns, will end up facing the same direction in which it started. The amount of each individual turn will be 360 divided by the number of turns, or 360/:N. (Now is the time to recall Logo's mathematical capacities. Review the Logo notation to add, subtract, multiply, and divide.)

You are ready to write the new procedure NGON:

```
TO NGON :N :EDGE
 REPEAT :N [FD :EDGE RT 360/:N]
END
```

Try it out. Notice that when :N becomes large, the drawn figure becomes a circle (almost). Carry out some clever visual experiments with NGONs.

Some observations

Look back carefully at what we have done so far with procedure writing. We started with a list of commands that drew a box of a single size. Next, we grouped these commands into procedures that could draw boxes of several different sizes. Next, we generalized the BOX procedure with an argument so that it could draw boxes of any size. Finally, we produced a still more general procedure, NGON, that can draw any regular, polygonal "box"—triangles, squares, pentagons, hexagons, and so on—of whatever size we wanted.

Making the simple more complete

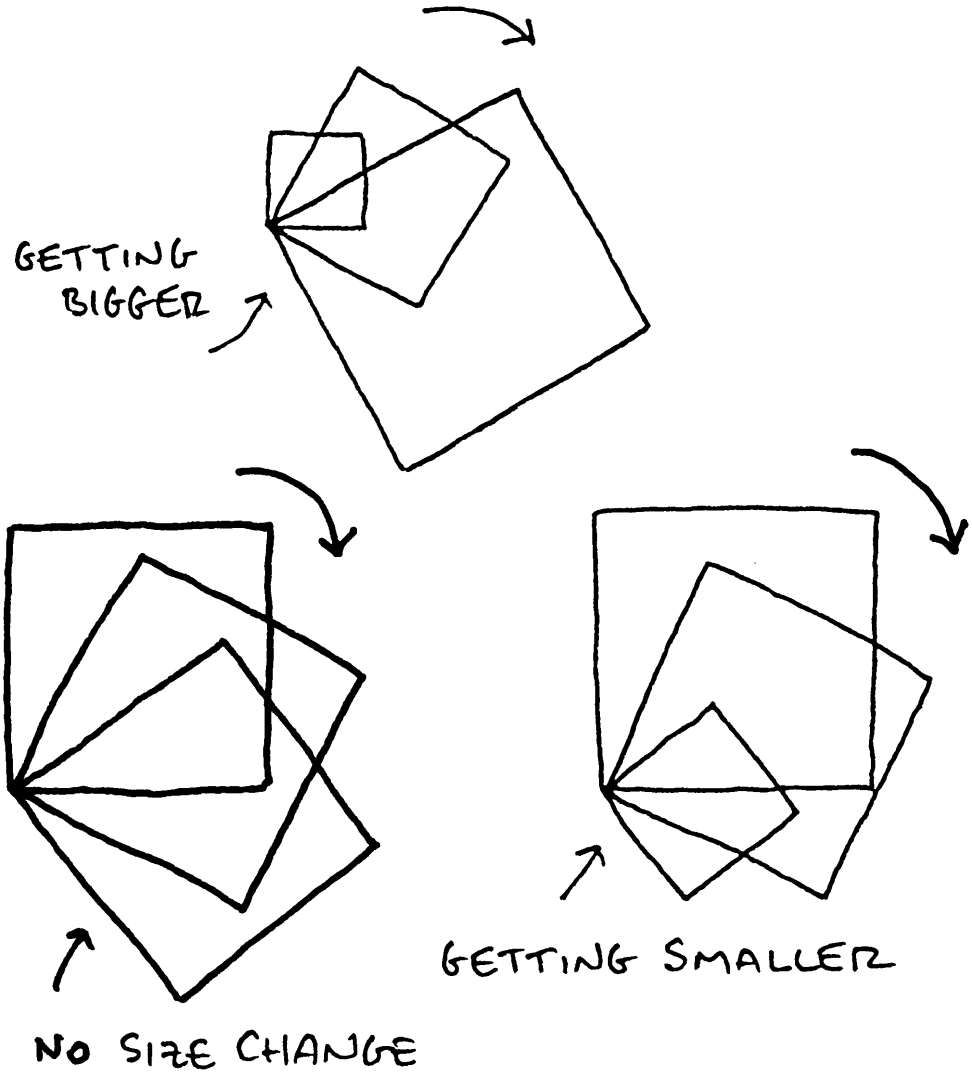
What next? How can we make these simple polygons more interesting? Maybe we can add another polygon characteristic to NGON.

Sometime in your life you were probably given an exercise like the following. "Cut out some different sized squares from a sheet of colored paper and think about placing the squares on a large piece of white paper. First, try to arrange the squares so that your design feels unbalanced and looks wonky. Next, rearrange the squares to create a balanced design."

What kinds of changes did you make? What about creating an arrangement that looks sad and another that looks euphoric? Each design used the same squares. What made them different? Their placement.

Make NGON draw a bunch of NGONs and put them on the screen according to some placement rule. Take out your notebook and doodle. Here are some results from my doodling. Your approach will be different. I'll explain mine, and I'll expect that your plans will go into your notebook.

Sketches of spinning polygons that grow or shrink as they spin



Word description of sketched ideas

"I want Logo to draw a series of polygons rotating around a common point, and I want each successive polygon to get bigger, or maybe get smaller.

"I don't know what angle to turn between one polygon and the next, so I'll include an argument called :ANGLE that I can vary, to see what happens.

"I don't know how big the growth should be between one polygon and the next, so I'll define another argument called :GROWTH. I'll play with different values of :GROWTH to see what looks best.

"How do I make :GROWTH work? Growth can be of two sorts: growth by a constant amount, or growth by a constant percentage. I'll try the latter. That means that if I want polygons to grow by 10%, I define :GROWTH to be 1.10. For a 90% shrinkage I would use :GROWTH = .9."

A procedure to spin polygons

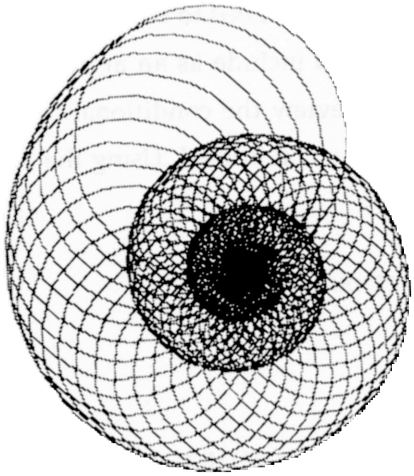
```
TO SPINGON :N :EDGE :ANGLE :GROWTH
 NGON :N :EDGE
 RT :ANGLE
 SPINGON :N (:EDGE*:GROWTH) :ANGLE :GROWTH
 ; Here is the recursion.
END
```

What is new here? First, there are more arguments than you have seen before. Every time you use SPINGON, you must remember to type four numbers after it. Second, this procedure is *recursive*: the last line in the SPINGON procedure asks that SPINGON be done again, but with some arguments changed. For example, (:EDGE) becomes (:EDGE*:GROWTH) the first time recursion is called; and then (:EDGE*:GROWTH) becomes (:EDGE*:GROWTH)*:GROWTH the second time recursion is called. A recursive procedure is a procedure that uses itself as one of its parts.

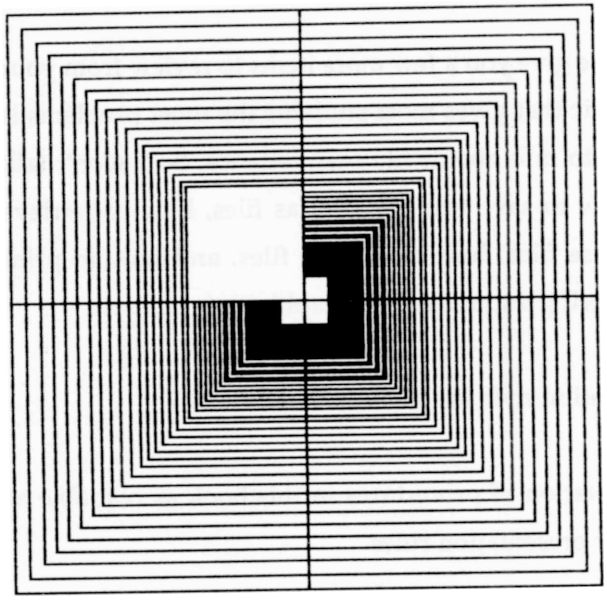
Will SPINGON ever stop? Try it out.

Some spingons

```
SPINGON 30 2 10 1.02 95
```



```
HOME CG REPEAT 3 [SPINGON 4 120 0 .95 50 RT 90]
SPINGON 4 120 0 .95 19
```



Stopping recursive procedures

One last procedural writing point to review. Having put SPINGON into motion, how do you make it stop at a stage of your choosing? You need to have a way of telling it how to stop. That's another characteristic to include as an argument. Look at the following modification to SPINGON. Review the conditional commands in Logo. IF . . . [something] is such a conditional. Using the IF phrase, everything becomes very tidy.

```
TO SPINGON :N :EDGE :ANGLE :GROWTH :TIMES
 ; Note the new argument above.
 IF :TIMES < 1 [STOP]
 ; This is the conditional stopper.
 NGON :N :EDGE
 RT :ANGLE
 SPINGON :N (:EDGE*:GROWTH) :ANGLE :GROWTH (:TIMES-1)
 ; Note the new argument above.
END
```

Boring logistics

Before ending this chapter, let me give you a few more items to review from your Logo language manual. The topic that gives most students the most problems is one of the most boring things to talk about: file maintenance. So I won't talk about it. But please review how to save text material as files, how to retrieve material from files, how to erase files, how to catalog files, and how to print files. Do the same review for storing and retrieving graphics information.

A note on the procedure presentation style used in this book

I have tried to make the presentation of procedures in this book as readable as possible. Here are several of my presentation rules.

First, the *many comments* rule. I have included wordy explanations in some of my procedures. These comments begin with the Logo command ";". There is, of course, no need for you to include these comments in your own version of my procedures. However, it is a good idea for you to put comments in your own procedures.

Second, the *meaningful cluster* rule. I often include extra parentheses to group like elements into a cluster. This is useful, for example, when an argument is composed of a collection of Logo material, but you want to see it as a single cluster of information. Here is an example from this chapter. Notice the use of comments, too.

```
TO SPINGON :N :EDGE :ANGLE :GROWTH :TIMES
 ; Note the new argument above.
 IF :TIMES < 1 [STOP]
 ; This is the conditional stopper.
 NGON :N :EDGE
 RT :ANGLE
 SPINGON :N (:EDGE*:GROWTH) :ANGLE :GROWTH (:TIMES-1)
 ; Note the new argument above.
END
```

Third, the *body structuring* rule. Procedures should be laid out nicely on the page without too much information on any one line. Long procedure statements should be divided up between lines to make them more readable. The special character "-" is used to indicate when a single Logo statement has been continued from one line to the next. Here is an example. Notice that the Logo material within the [repeat brackets] would have been difficult to read if the long statement had not been divided into several short lines.

```
TO SQUIGGLE :A :B :N
 REPEAT :N [ FD :A      -
             RT 130     -
             FD :A      -
             RT 50      -
             BK :B ]
END
```

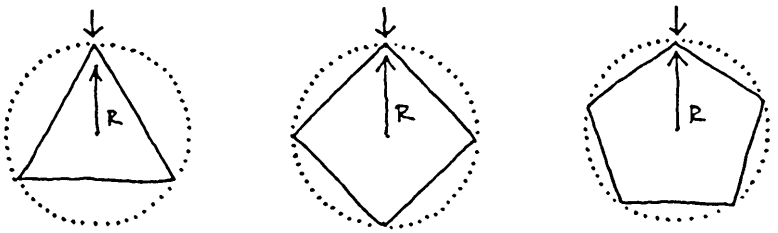
The symbol "–" indicates, of course, that the return key should not be used because the Logo statement continues. Consult your own Logo manual for handling the problem of procedure layout.

Exercises

There are five exercises to explore before going on to the material of chapter 2. The first is so important that we will go through it together, step by step. You can work on the other exercises by yourself.

Exercise 1.1

Make NGON more versatile by doing two things to it. First, improve NGON so that it will draw polygons *around a central point;* and second, improve NGON so that it can be given an argument that specifies not the length of an edge of the polygon but the *radius of the polygon.* I've made up the term radius of a polygon. It is the radius of the smallest circle that just encloses a regular polygon. The center of this circle is the point around which the polygon is to be drawn. See the diagram below.



Call the revised procedure CNGON for "centered NGON."

Two hints: First, ask yourself what arguments CNGON will need. This is another way to ask yourself what information must be given to CNGON so that it can go about its business of drawing centered NGONs. CNGON needs only two pieces of information, or two arguments: the number of sides of the polygon to be drawn and the radius of that polygon. That means that the first line of the new procedure will look like this:

```
CNGON :N :RAD
```

Second, imagine yourself as the turtle. How would you walk through the design that CNGON must make? Draw a simple diagram to describe such a "turtle walk." You might want to divide the diagram up into individual scenes. Later you can translate each scene in words and then into Logo notation. Here is the first instance where this turtle visualization is really needed. Let yourself go; talk out loud; get on with it without too much thinking.

Word description of the turtle walk (see sketches on next page)

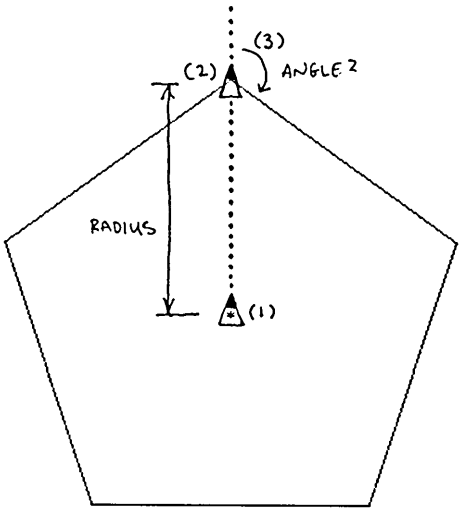
Diagram A: Getting ready to draw the polygon.

You, as the turtle, begin your journey from position (1), the center of the proposed polygon. You are facing straight up.

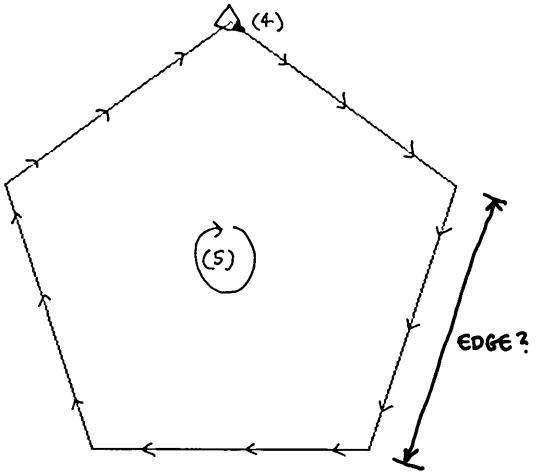
Pick up your pen and move forward by the amount of the polygon's radius. This is :RAD. This puts you in position (2).

You now need to turn right by an amount that is labeled (angle) on the sketch (3). What will (angle) be related to? Will (angle) be different for differently shaped polygons, that is, polygons with different numbers of sides? Yes. Will (angle) be related to the overall size of polygons? No. Don't worry about how to calculate (angle), yet; you can work that out later.

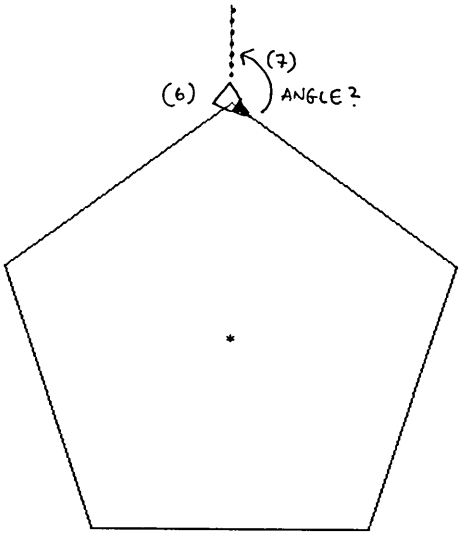
Turtle walk sketches



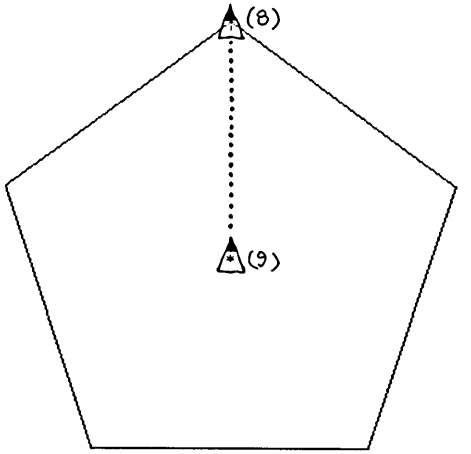
A



B



C



D

After turning right by the amount of (angle), you are looking along one of the polygon's edges (4).

Put down the pen in preparation for drawing the polygon.

Diagram B: Drawing the polygon.

You are at position (4) and ready to draw an n-sided polygon. You can use the procedure NGON. But what arguments will you use? It needs some values for :N and :EDGE. Right?

Yes, but wait a minute. What should the value for :EDGE be? You know the value of :N, the number of sides of the polygon. And you know the value of the new argument, :RAD. What must the polygon's edge dimension be so that, after it is drawn, it has a radius equal to :RAD? In other words, we need to be able to express :EDGE in terms of :N and :RAD. OK. We know the problem, what we have to work on, but let's not stop yet. Label the edge thing that must be calculated (edge). We will return to it in a minute.

Now you can draw an NGON :N (edge).

You began the NGON from position (4). You will end at the same place.

Diagram C: Getting ready to return to the center.

You are at position (6). You must now turn left by the amount of (angle); this is indicated by (7). This leaves you in position (8), pointing straight up.

Diagram D: Returning to the center of the polygon.

Pick up your pen and back down, by an amount equal to :RAD, to the polygon's center. Finally, put your pen down in preparation for whatever might come next. Note that you, as the turtle, have ended in the same position (9) as you began (1).

A turtle walk transformed into a Logo procedure (almost)

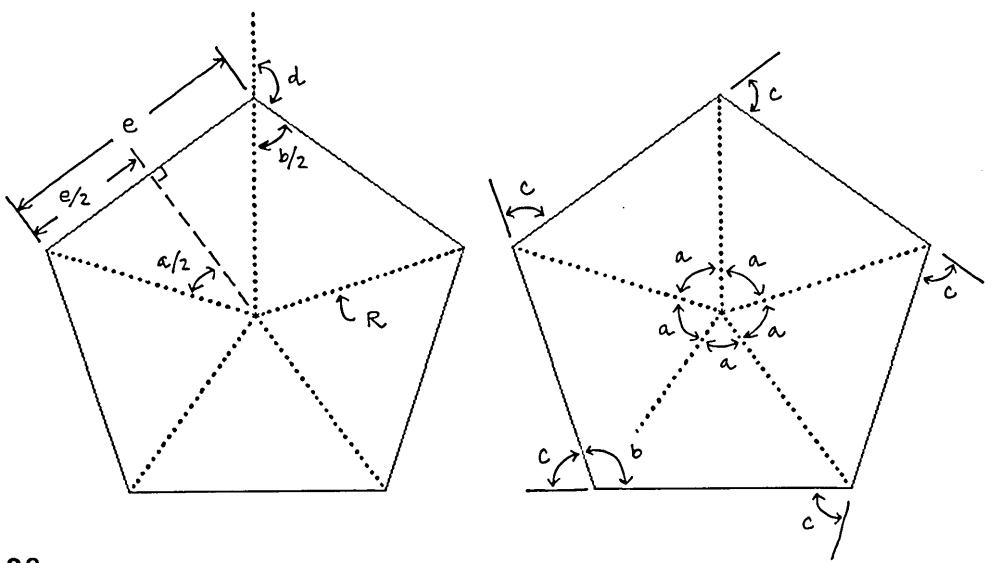
No more words are necessary. Here it is.

```
TO CNGON :N :RAD
 PU FD :RAD
 RT (angle) PD
 NGON :N (edge)
 LT (angle)
 PU BK :RAD PD
END
```

The procedure is sketched. And we know what we know and what we don't. The two amounts, (angle) and (edge), are still unknown. To figure these bits will require a little geometry and trigonometry. We might as well use this opportunity to review all the bits and pieces of polygons.

The Geometry of CNGONs

Use the following two diagrams in conjunction with the word and equation descriptions.



The first problem we face is to find an expression for angle d in terms of n, the number of sides of the polygon. The second problem is to find an expression for the length of a polygon's edge, e, in terms of its radius, R, and n, the number of sides.

In preparation for these two acts, let's look at all the angles associated with polygons. The *central angles*, labeled a, are easy. They are each equal to 360/n. The *external angles*, labeled c, are also equal to 360/n. The external angle is the turning angle used in NGON. What about the internal angles, labeled b. We need some work here:

(1) c = 360/n,
(2) c + b = 180.

Putting these two equations together and solving for b gives

(3) b =180*(n - 2)/n.

We are now ready to handle the first problem: Find d in terms of n.

(4) d + b/2 = 180.

Putting (3) and (4) together and solving for d gives

(5) d = 180 - 90*(n - 2)/n <---First problem solved.

OK, now look at the second problem: Find e in terms of R and n.

(6) sin (a/2) = (e/2)/R,
(7) a = 360/n.

Putting (7) and (6) together and solving for e gives

(8) e = 2*R*sin (180/n) <---Second problem solved.

Review the following trig functions: sine, cosine, and tangent. What is an arctangent? Draw some diagrams to explain each of these functions. Glue them on the inside cover of your notebook.

Installing the necessary geometry into CNGON

Here's how far we have gotten with CNGON:

```
TO CNGON :N :RAD
 PU FD :RAD
 RT (angle) PD
 NGON :N (edge)
 LT (angle)
 PU BK :RAD PD
END
```

Now we can replace `(angle)` and `(edge)` with the needed expressions. Here is the finished CNGON:

```
TO CNGON :N :RAD
 PU FD :RAD
 RT 180 - (90*(:N-2)/:N) PD
 NGON :N (2*:RAD*SIN (180/:N))
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
END
```

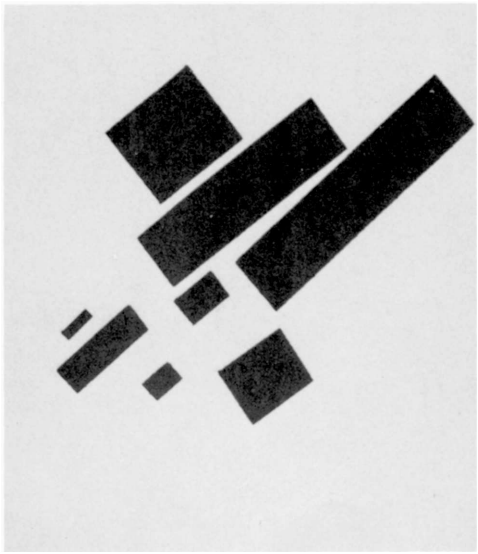
Lessons and tips

When solving visual problems, like this CNGON thing, try to break the single big problem down into several smaller problems. Solve each of the small problems in turn, and then plug the little solutions together to form one big solution.

Exercise 1.2

Put together one or more DEMO procedures that make imaginative use of the ideas presented and reviewed in this chapter. Modify every procedure in the chapter. Make them act more strangely.

You might want to think some more about the exercise, described above, of placing simple shapes on a blank field of paper to depict different feelings of balance or emotions. A painting by the Russian artist Kasimir Malevich is reproduced below. What could be more elegant that these eight red rectangles? (That's the title, by the way.) What emotion do you feel when looking at this little reproduction? Can you do something similar?

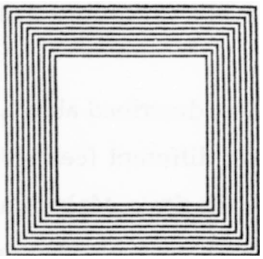


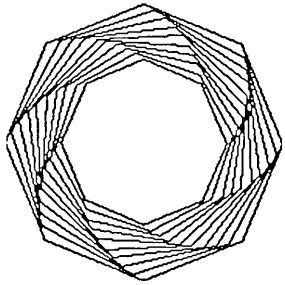
Exercise 1.3

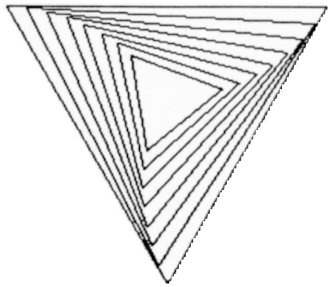
Combine the ideas of `SPINGON` with your newly constructed `CNGON`.

Exercise 1.4

Design a fancier `CNGON` that fills up a polygon with textures. Here are a few visual tips for Exercises 1.3 and 1.4:







Exercise 1.5

Design a Logo procedure that puts polygons on the vertices of other polygons. Hint: look carefully at the body of CNGON. When does the turtle arrive at a polygon vertex? Mark the vertex arrival place in the CNGON procedure. You might consider this location as the right spot to install some recursion: when the turtle arrives at any vertex, ask it to do another CNGON centered on that vertex. This recursive drawing will place polygons on the vertices of polygons on the vertices of polygons. You will need to figure out a way of stopping the recursion machinery, or it will continue forever. Define a new argument, :LEVEL, to keep track of recursion levels. What about the relative sizes of the polygons? Should they get bigger or smaller? Can you handle that?

Don't forget the following tricks:

1. Imagine yourself as the turtle.
2. Sketch the design that the turtle must walk through.
3. Describe the turtle walk in words.
4. Divide the walk up into logical sections.
5. Note the bits you know and the bits that will need some further thought and research.
6. Translate the words into Logo commands.
7. Test it out with realistic and totally outlandish argument values.

Chapter 2
Visual Modeling

"In the case of miniatures, in contrast to what happens when we try to understand an object or living creature of real dimensions, knowledge of the whole precedes knowledge of the parts. And even if this is an illusion, the point of the procedure is to create or sustain the illusion, which gratifies the intelligence and gives rise to a sense of pleasure which can already be called aesthetic."
Claude Lévi-Strauss, *The Savage Mind*

Models

Most physical models are miniatures, smaller versions of something else. Model railroad engines are good examples, and many of us have had pleasant experiences with them. They can be picked up and looked at from any angle, and they can be experimented with, too. How many wagons, for example, can a model locomotive pull? Attach 10 wagons and see if the locomotive can pull them. And if you should sit beneath a model railway bridge when the tiny engine rolls across it, pulling all those wagons, what will the sound be like? Will it be like the real thing? Listen carefully and you will experience a double thrill: an excitement that comes from using a model to hear how a full-sized locomotive might sound; an excitement that comes from simply playing, on your own terms, with a miniaturized piece of the world.

I believe that this kind of play, because it encourages us to look more closely at our world, is very useful enjoyment. In addition, I am convinced that the clarity of vision developed by such play is best pursued by involving ourselves, not in just the manipulation of models but in their design and construction as well.

This book is about a special kind of modeling that explores patterns and visual images. Sometimes we will create designs using visual models of machines that are very much from the real world. Other times, images will be produced by more abstract or imaginary machinery. All our models though, whatever they represent, will be constructed from Logo procedures. Logo is the raw material, but this book is not *about* Logo. In fact, you must try hard not to let Logo get in the way of your model building craft. Models first. OK?

OK. No more introduction. Let's build an image-producing machine using Logo. But first . . .

An important pause in the narrative

What comes next requires that you have a fair understanding of the Logo language. But what, you ask, does *fair* mean? If you haven't yet looked at the material in Chapter 1, now is the time. Chapter 1 will let you compare your current knowledge of Logo mechanics with what you will need to build the visual model described in the next section. Don't just scan Chapter 1; go through the examples in the text, and try out the exercises at the chapter's end.

Even if you already know some Logo, a review probably would be helpful. Chapter 1 describes much of the geometry and trigonometry used throughout the book, so now is a good time to review that material. Try to apply some geometry to a few of my exercises. In addition, Chapter 1 will introduce you to my Logo-talking style; maybe you should get used to that style right away. Be sure to spend some time working on the centered polygon assignment. This exercise combines a review of Logo, the introduction of turtle walks, and simple geometry.

Go back and take at peek at Chapter 1 right now. Have a good read and take your time with it. If you are a real Logo high-flier, test your flair by doing the exercises at the end of Chapter 1, before you glance through the hints given there. After having a go at these problems, compare your Logo style with mine.

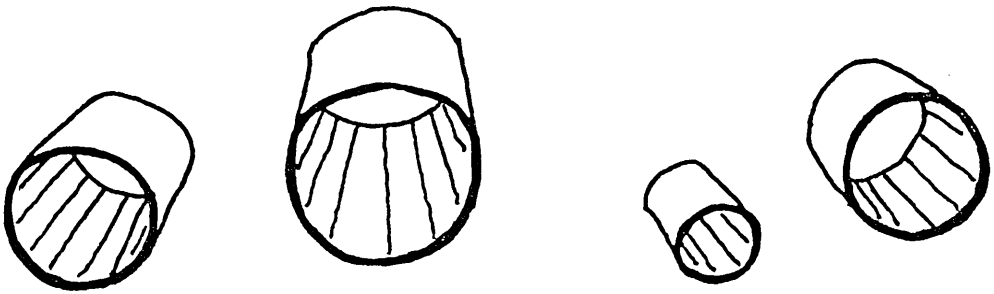
On to modeling

I have decided to begin this work with something concrete, something selected from the world of things. We can do some abstract modeling of ideas or emotions later on. The example that follows is a description of a device that I vaguely remember seeing described in an old *Scientific American* magazine. I've called it a pipe-and-roller machine. I never built the thing, but I always wanted to *see* how it worked.

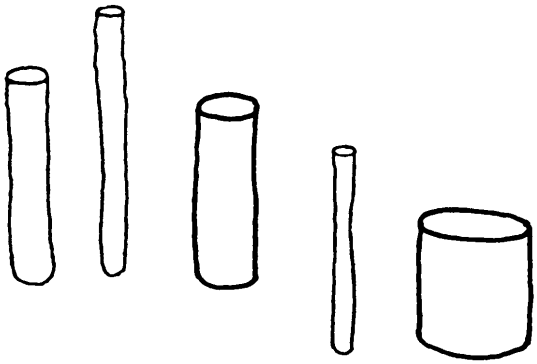
Pipes and rollers

Imagine that we are wandering about a construction site where there are useful bits and pieces lying about, free for the taking. While this wandering will be done only in our minds, the images seen there are based on impressions from past, real excursions.

We will need some short sections of plastic pipe, the kind used for plumbing. Try to find as many different-diametered pipes as you can. Here is a sketch of what you might have picked up so far.



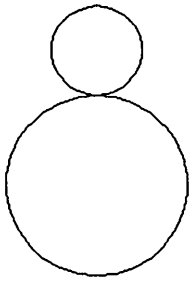
Next, let's assemble a collection of wooden dowels of various diameters, from very small diameters to very large ones. Dowels, or rounded wooden pegs, are used to join together adjacent parts by fitting tightly into two corresponding holes. Dowels are used by cabinetmakers to assemble fine pieces of furniture when nails or screws would be unsightly; and large dowel-pegs are still used to fit together wooden beams when aesthetics are more important than cost. Call the dowels that you have assembled "rollers"; you will see why in just a minute. Here is a sketch of my dowel collection. (Sketches are models, too.)



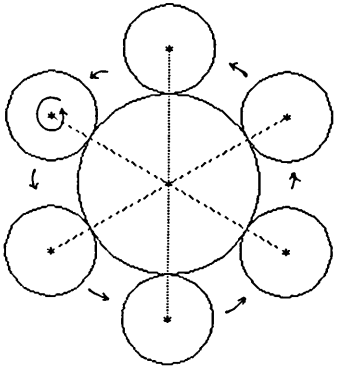
Assembling the pipe-and-roller machine

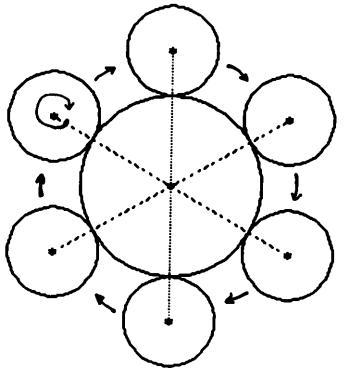
Now, imagine that one section of pipe is floating in the air at eye level; one end of the pipe is clearly visible to us, and the pipe's length is parallel to the

ground. Now, hold one of the dowel-rollers parallel to the length of the pipe and place it on top of the pipe.



Next, imagine rolling the dowel around the circumference of the pipe until it arrives back at its starting position at the top of the pipe. You will have to hold the roller very carefully so that it doesn't slip on the pipe but rolls nicely in contact with the pipe. If you roll the dowel around the pipe in a counterclockwise direction, the dowel will also turn counterclockwise.

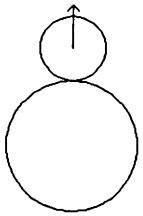




The dowel would turn in a clockwise direction if the rolling-about-the-pipe was also in a clockwise direction. Note the two motions of the roller: the roller goes around the pipe as it turns around it own center. The two centers, the center of the pipe and the center of the roller, will feature in all our calculations.

Now that you have an image of the physical machine in your mind's eye, I can ask you to begin manipulating the parts of the image. Imagine playing with this model in your mind. What will the roller "look like" as it rolls around the pipe? Can you draw a picture of it? Or better still, can you create a Logo model of this roller/pipe machine that can *illustrate* the motions for us?

Imagine, for example, that we glue an arrow onto the end of the roller.



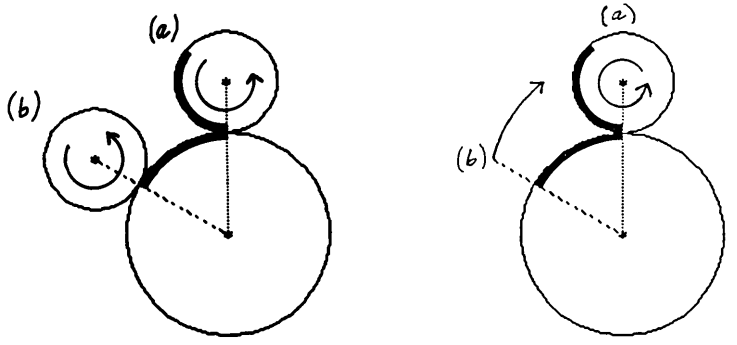
What pattern will the tip of this arrow trace out as the roller moves around the pipe? Imagine a series of photographs taken at regular intervals as the roller moves around the pipe. Let's construct a Logo machine that will work in this photographic way. In other words, let's build a Logo machine to *model* the physical pipe-and-roller events *visually* .

Roller talk

At first glance, this exercise looks pretty difficult, certainly more complicated than the centered polygon problem discussed in Chapter 1. But, if we could just break this problem down into smaller, more manageable parts, as we broke the polygon problem down, some of the complexity might vanish. So let's set about doing just that.

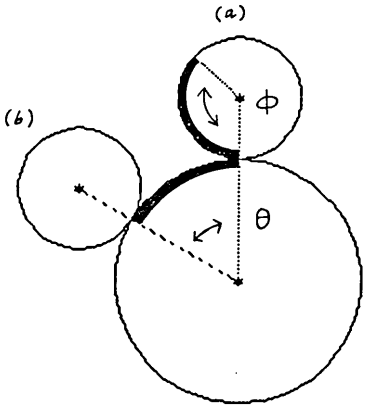
First, we can talk about the geometry of rolling cylinders to see what we already know. Then, we can do a turtle-walk scenario—with sketches and words. Finally, Logo will act as the glue to stick all the individual parts together.

Look at the figure on the left below. We see the roller at the top of the pipe, position (a), and then rolled counterclockwise around the pipe to position (b). How much has the roller turned between points (a) and (b)? The dark bands in the figure indicate the contact surfaces between the roller and the pipe. If there is no slipping, the length of the band on the roller must equal the length of the band of the pipe. Why?



We could think of this rolling in another way. See the figure on the right above. Imagine that the roller *stays* fixed at position (a). The roller now rotates at (a) while the pipe is *rotated*—clockwise—underneath it from (b) to (a). In this alternative view, the roller also turns in a counterclockwise manner, by the distance indicated by the dark band. Here, as in the figure on the left, the length of the band on the pipe must equal the length of the band on the roller.

How can we calculate the lengths of these bands? First, note that the bands can be described as segments of a circle, that is, some fraction of the total circumference of a circle. Well, what do we know about circumferences? Any circle's circumference, C, equals $2\pi R$, where R is the radius of the circle. Now look at the figure on the next page.



The band on the pipe is some fraction of the circumference of the pipe. This fraction is the angle θ (theta), measured in degrees, divided by 360, the total number of degrees in a circle. The length of the band on the pipe is, therefore, $2\pi R_p\theta/360$, where R_p is the radius of the pipe. The same thinking provides the length of the band on the roller: $2\pi R_r\phi/360$. Here, R_r is the radius of the roller.

What next? We can set these two expressions equal to each other, since the physical dimensions they represent are equal to each other in length. Then we can rearrange terms to express φ (phi), the degree rotation of the roller, in terms of θ, the degree distance between (a) and (b). Here it is:

$$\phi = \theta R_p/R_r.$$

This rotation expression is very convenient. If we know how many times we want to photograph the roller on its way around the pipe, we can calculate θ, the degree distance between stoppings, by dividing 360 by the number of stoppings. And knowing the radius of the pipe and of the roller, we can use the tidy expression above to calculate φ, the relative rotation of the roller from one stopping point to the next.

So much for the roller talk. But before we go on to the turtle walk, will you admit that you know more about this problem than you thought you knew at the outset? Listen: breaking big problems down into smaller ones makes getting

started easier. And once you get started moving in any direction, you will discover that you are already familiar with much of the scenery.

A turtle walk around the pipe

Remember that a turtle-walk scenario describes in words and sketches how you want the turtle to walk through a design. Let yourself go, but be specific. Addressing your instructions to the turtle and talking out loud may be helpful. Let's use the sketches on the next page as the focal point of this scenario. I have divided my turtle walk into small scenes and have given each a letter designation.

The previous shapes in this section were drawn with Logo procedures, but I have intentionally left the following figures in *freehand* form; they are taken from my own Logo notebook. I wanted to remind you that sketches come *before* Logo procedures that draw rounder circles. The following sketches record my visual doodling about this particular problem. But to appreciate the usefulness of sketches, you must do some yourself. Don't just look at my examples.

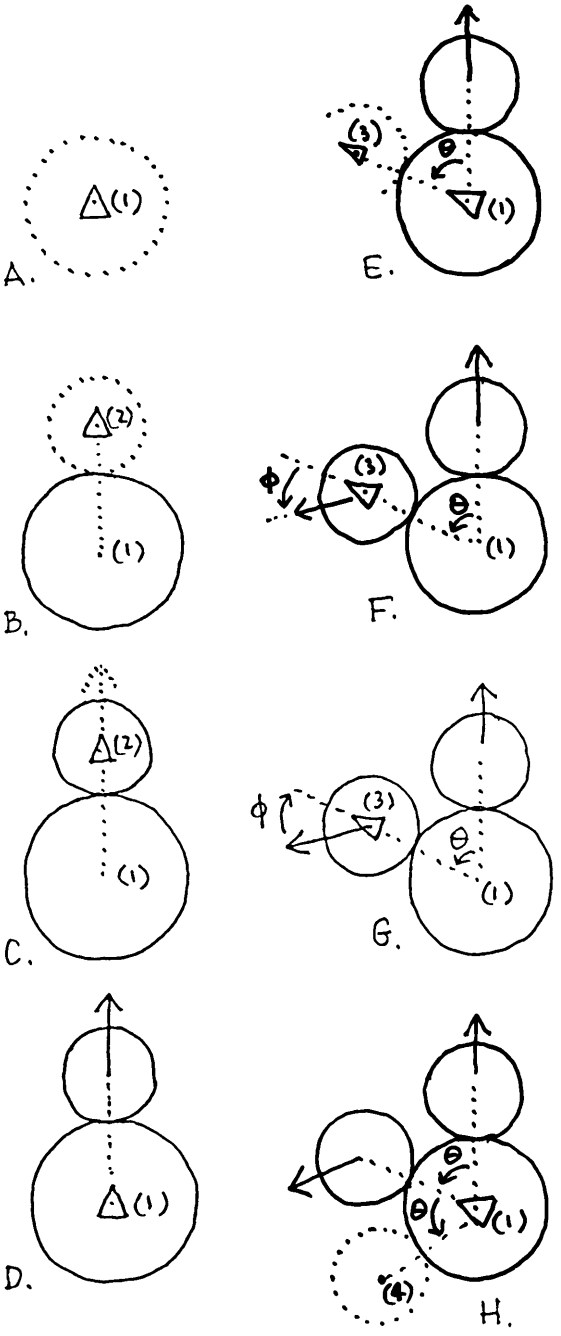
Because sketches can be effective visual aids for careful thinking, they need to be drawn carefully. I occasionally use rulers and a compass, but not always. Of course, the small diagrams on the next page are final sketches, not beginning ones. Final drawings, like final Logo procedures, are the results of many preliminary studies, many of which did not "work out properly."

Word description of the turtle walk

Diagram A: Drawing the pipe

Begin at position (1) facing straight up. Draw a circle around point (1) with radius R_p. This will be easy to do using CNGON. (This procedure is listed below, but see Chapter 1 for a full description of it. Think of it as a black-box procedure

Turtle-walk sketches of the roller-pipe machine



that draws regular polygons around a central point. CNGON takes two arguments: N, the number of sides of the polygon to be drawn, and RAD, the radius of the circle that circumscribes the polygon. For example: CNGON 20 60 would draw a circle—a 20-sided polygon—of radius 60. The turtle's current position defines the *center* around which the polygon would be drawn.)

Diagram B: Drawing the roller in its first position on the top of the pipe

Next, pick the pen up and move to position (2), the center of the roller. This distance is $R_p + R_r$. Now draw a circle of radius R_r centered on position (2) to illustrate the roller.

Diagram C: Orienting the roller in preparation for drawing the arrow

Since the roller has not yet moved from the starting position, it hasn't done any rolling. Hence, the arrow can be drawn pointing straight up, and that is the direction in which you are facing. Draw the arrow starting from position (2) and get back to this position when you are finished.

Diagram D: Getting back to the center of the pipe

Pick up the pen and move back to the center of the pipe, position (1).

Diagram E: Getting to the next stopping position of the roller

Turn left by angle θ, and go forward to position (3). At point (3) you must correctly orient the roller before drawing the arrow. Because the roller has now rolled a bit to the left of its starting position at (2), the arrow will no longer point in the same direction as the line: (1) to (3). It will be turned some amount to the left of it. What must you take into account to calculate the rotation amount?

Diagram F: Orienting the roller and drawing the arrow

The roller has moved from position (2) to position (3) by rotating about its own center. We used the symbol ϕ to indicate this rotation. The angle ϕ is measured relative to the dotted line linking the centers of the pipe and roller: (1) to (3).

You have arrived at position (3), pointing along the axis (1) to (3). If you now turn left by angle $\phi = \theta R_p/R_r$, you will be facing in the correct direction to draw the arrow. Draw the roller circle, too.

Diagram G: Getting back to the center of the pipe

Turn right by ϕ, pick up the pen, and move back down to (1).

Diagram H: Preparing for the next roller stopping position

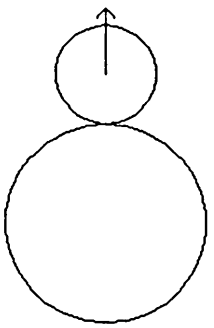
Get ready to draw the next roller image: turn left by angle θ and move out to position (4). The roller rotation angle at the point (4) is again measured *relative* to the dotted line linking points (1) to (4). Why?

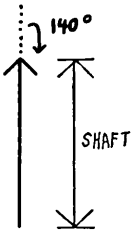
Angle ϕ at position (4) equals $2\theta R_p/R_r$. Why 2θ? Because ϕ must be calculated relative to the starting position, and the roller has moved 2θ degrees from the starting position (2). Turtle: you may now turn left by ϕ, draw the arrow, turn right by ϕ, and go back down to the center of the pipe.

Angle ϕ at the next stopping position (5) is not shown in the diagrams. But it will be $3\theta R_p/R_r$. Why? Draw a few diagrams to convince yourself of all this. Go back and look at the figures on page 40 for some help.

A turtle walk transformed into Logo procedures

To start, recall that we have to glue an arrow onto the face of the roller. So let's write a procedure to draw an arrow of any shaft length `:L`.






```
TO ARROW :L
 ; To draw a simple arrow of shaft length :L. The length of
 ; each tip is given by .2*:L.
 ;  PD FD :L
 LT 140  FD .2*:L  BK .2*:L
 RT 280  FD .2*:L  BK .2*:L
 LT 140 BK :L
 PU
END
```

PIPEGONs

Let's call the procedure that will carry out this turtle walk PIPEGON. What will be the arguments? Certainly the radius of the pipe and the radius of the roller will be needed. We will also need to know θ and how many stopping points we would like to photograph. Here is the list of arguments so far:

:RP, the radius of the pipe
:RR, the radius of the roller
:THETA, the angle distance between stopping places
:N, the number of stopping places

Let's add one more, :CUM, that will keep track of the total of the angle turned from the starting roller position. We can now write the first line of PIPEGON:

```
TO PIPEGON :RP :RR :THETA :CUM :N
```

How do you feel about rushing right into doing the rest? The following is not my first "rush" or even the second. My first few attempts had bugs in them, and they didn't work as I had planned. But procedures almost never work the first time. That's OK as long as your energy is up to fixing them.

```
TO PIPEGON :RP :RR :THETA :CUM :N
 IF :N < 1 [CNGON 20 :RP STOP]
 PU FD :RP + :RR PD
 LT :CUM*:RP/:RR
 ARROW :RR*1.5
 CNGON 20 :RR
 RT :CUM*:RP/:RR
 PU BK :RP + :RR
 LT :THETA
 PIPEGON :RP :RR :THETA  (:CUM+:THETA)  (:N-1)
END
```

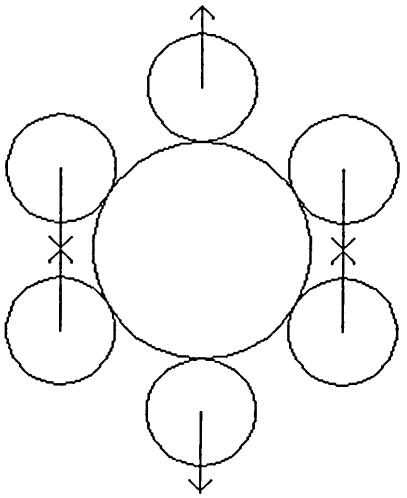
Supporting procedures

```
TO CNGON :N :RAD
 ; To draw an N-sided polygon centered on the turtle's
 ; current position. RAD is the radius of the circle that
 ; would pass through all of the polygon's vertices.
 ; See Chapter 1 for a full description of CNGON.
 PU FD :RAD
 RT 180 - (90*(:N-2)/:N) PD
 NGON :N (2*:RAD*SIN (180/:N))
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
END

TO NGON :N :EDGE
 ; To draw an N-sided polygon. The first edge will be
 ; drawn from the turtle's current position, and its length
 ; is given by EDGE.
 REPEAT :N [FD :EDGE RT 360/:N]
END
```

Some pipegon productions

I typed `PIPEGON 60 30 60 0 6`. This models, in a visual way, the rolling of a

roller of radius 30 around the circumference of a 60 radius pipe. The roller stops along the circumference every 60 degrees, and 6 rollers will be drawn. The argument :CUM is given an initial value of 0. What is :CUM being used for? What happens if you begin with some other value, say 43.5?



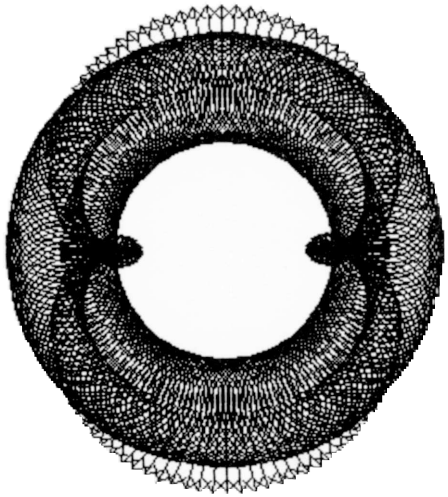
One last point. In my turtle-walk scenario I drew the pipe circle before doing anything else. The procedure PIPEGON draws it last. Why did I change the order of things? Well, I wanted to use recursion and to be able to specify the number of times recursion would happen. I used the argument :N to take care of this. PIPEGON's first line looks at the current value of :N; when :N becomes zero, PIPEGON should be stopped. It is easier to know when a procedure should be stopped than when it has just begun, and this seemed a nice place to draw the pipe, after *all* the rollers had been drawn. Could you reorganize the procedure to draw the pipe before drawing any of the rollers?

Exploring PIPEGON dynamics

One of the pleasures of modeling is playing with the little model you have

built. Let's fiddle with PIPEGON's parts to *see* what happens. I will show you only a few things to give you the idea. Let's start with some different argument values.

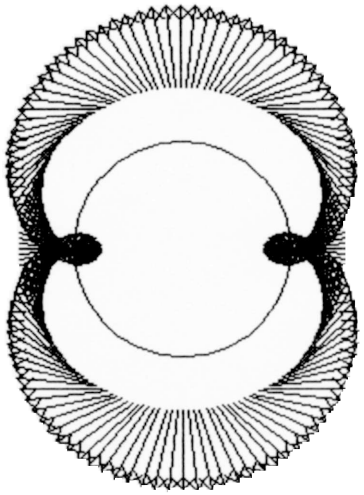
Here is the portrait of `PIPEGON 60 30 2 0 180`.



But I don't like all those circles. So I removed PIPEGON's third line. Here is the new version. The asterisks `(***)` mark where the line was removed from the original version of the procedure. Don't type them, though.

```
TO A.PIPEGON :RP :RR :THETA :CUM :N              <--- new name
 ; Arrow-only pipegon
 IF :N < 1 [CNGON 20 :RP STOP]
 PU FD :RP + :RR PD
 LT :CUM*:RP/:RR
 ARROW :RR*1.5
 (***)                                      <--- line removed
 RT :CUM*:RP/:RR
 PU BK :RP + :RR
 LT :THETA
 A.PIPEGON :RP :RR :THETA (:CUM+:THETA) (:N-1) <--- new name
END
```

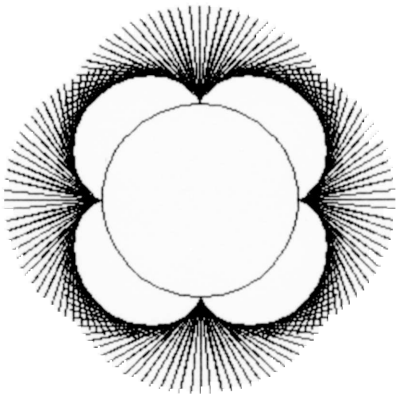
Now this is a portrait of A.PIPEGON 60 30 2 0 180.



Instead of drawing an arrow on the roller, let's draw a stripe along a diameter. We can use CNGON to draw a two-sided polygon with radius equal to the roller. We take out the ARROW procedure and insert CNGON. Here it is:

```
TO S.PIPEGON :RP :RR :THETA :CUM :N              <--- new name
 ; Striped pipegon
 IF :N < 1 [CNGON 20 :RP STOP]
 PU FD :RP + :RR PD
 LT :CUM*:RP/:RR
 (***)                      <--- ARROW removed
 CNGON 2 :RR                <--- 2-sided CNGON installed here
 RT :CUM*:RP/:RR
 PU BK :RP + :RR
 LT :THETA
 S.PIPEGON :RP :RR :THETA (:CUM+:THETA) (:N-1) <--- new name
END
```

And here is a portrait of `S.PIPEGON 60 30 2 0 180`.



Now, imagine an invisible arrow glued to the front of the roller. At the tip, there is a flashing light. Here is the new part to fit into our `PIPEGON` machine:

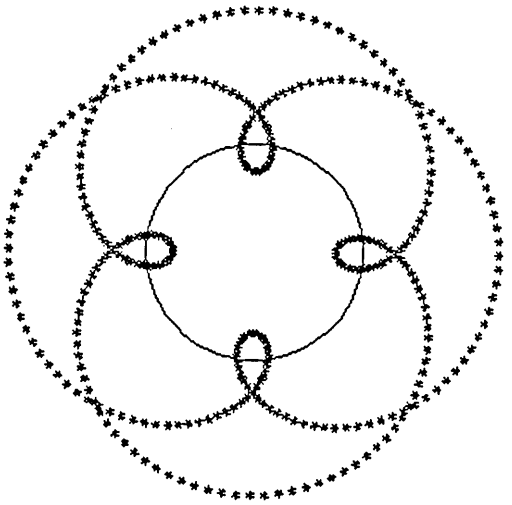
```
TO FLASH :L
 ; Flashes a light at distance :L from the starting point,
 ; and returns the turtle to where it started.
 PU FD :L PD
 REPEAT 6 [FD 2 BK 2 RT 60]
 PU BK :L PD
END
```

To install `FLASH` into `PIPEGON`, we could fix a value for `:L`, perhaps based on the value for `:RR`. Or we could extend `PIPEGON` by adding another argument. Call the extension `L.PIPEGON`.

```
TO L.PIPEGON :RP :RR :L :THETA :CUM :N <--- new name and arg
 IF :N < 1 [CNGON 20 :RP STOP]
 PU FD :RP + :RR PD
 LT :CUM*:RP/:RR
 FLASH :L                    <--- FLASH installed.
 (***)                       <--- CNGON removed
 RT :CUM*:RP/:RR
 PU BK :RP + :RR
 LT :THETA
 L.PIPEGON :RP :RR :L :THETA (:CUM+:THETA) (:N-1)
                                        <--- new name and arg
END
```

Here is the flash portrait of:

```
L.PIPEGON  60  30  40  2  0  180
L.PIPEGON  60  30  -40  2  0  180
```



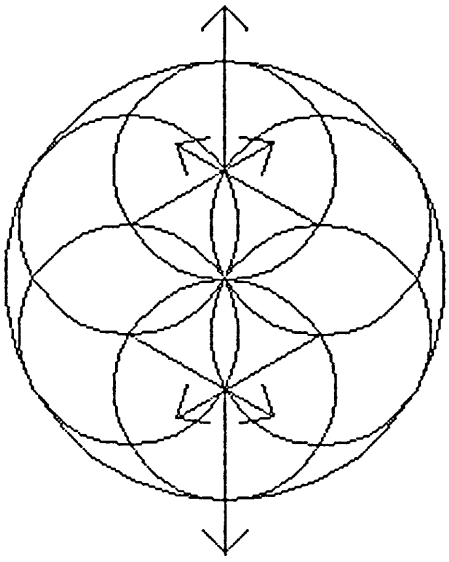
What happens if we make the radius of the roller negative? Right. The roller is inside the pipe. Some examples are shown on the next page.

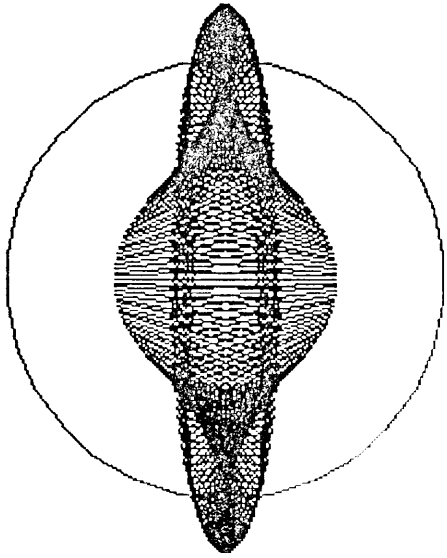
Closure

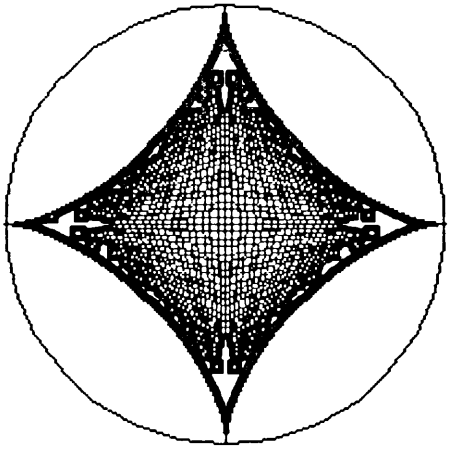
Have you noticed that even the most complex designs we have done so far are drawn quite quickly? Each is complete by the time the roller has made a single 360-degree trip around the pipe. If the roller makes a second trip around the pipe, the design repeats exactly. We can describe this kind of design as one that has "closed upon itself" or, more briefly, that has "closed" after one trip.

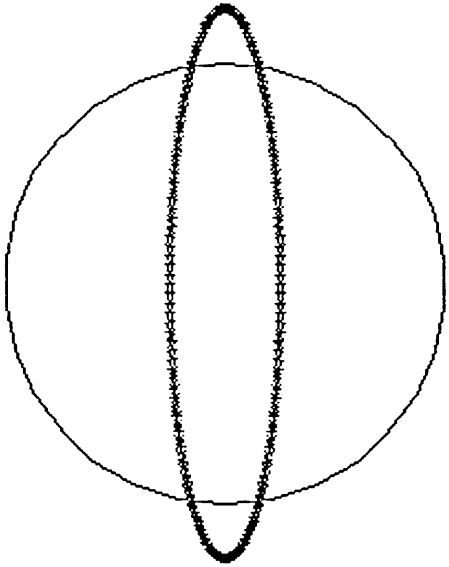
Not all designs produced by our `PIPEGON` machine will close after only one trip; some will take several trips to close, and others will require a great number of trips. Experiments will show that altering the sizes of the roller and pipe leads to different closure patterns. What determines the number of trips before closure occurs? Can you calculate the trips until closure if you know the sizes of the roller and pipe? Could you "find" a design that never closed?

A portfolio of roller-inside-pipe portraits









On the next page is a PIPEGON design that closes only after a number of trips around the pipe. The individual images show the design at various trip stages around the pipe. Can you guess the pipe and roller sizes I used?

Words elicit images

Words that are visually descriptive, like *closure*, should call up a variety of images in your mind. This elicitation of mind-images can be enormously useful in visual modeling. In each of the following exercises, I will stress the importance of words. We must talk a lot in conjunction with sketching a lot. Take a few minutes here to think visually about the word *closure*. Say "closure": what images does it bring to mind? Tell the turtle to "hurry up and bring a design to a close." Jot down, or *sketch*, the image ideas elicited in your own mind by the chanting of the word. Put it all in your notebook.

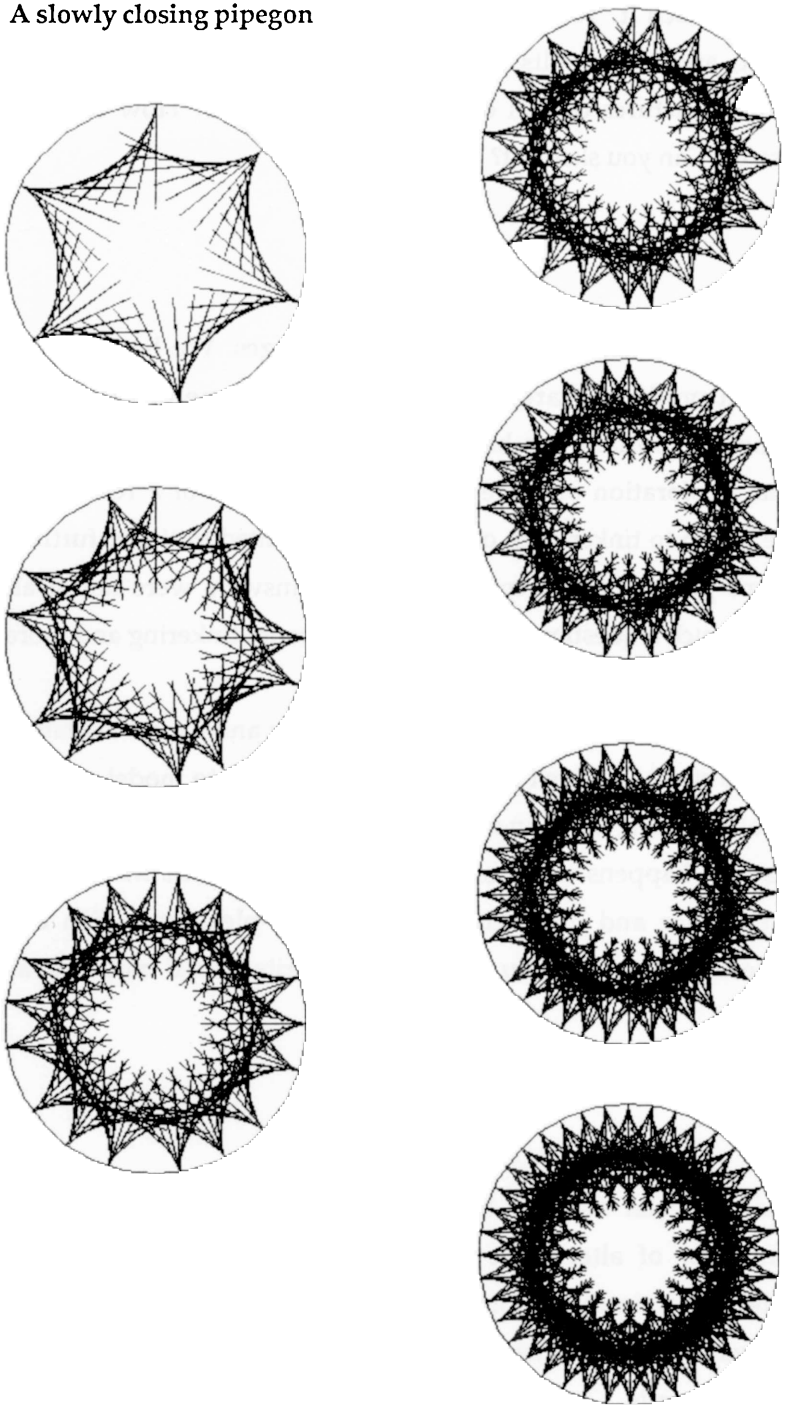
Suppose you needed to find a synonym for closure. What would you suggest? Any suggestion must be descriptive of all the image work we have completed. By the way, you probably won't find closure in a standard dictionary. Why is this?

Imaginary machines

At the start of this chapter I mentioned that sometimes we would model machines from the real world—pipes and rollers are very real world—and other times we would model machines that aren't so real. Perhaps we can make one model do both real and imaginary things. For example, can we make our PIPEGON machine draw some fantastic designs? (By the way, look at that word *imaginary*. Why does it have *image* in it? Can you imagine why?)

Let's imagine a striped roller inside a pipe. The procedure PIPEGON will generate a composite picture of this roller as it travels around the inside of a pipe. So far, this is just like the situations viewed above. But now, let's introduce the *fantasy* feature. Make the radius of the roller larger than the

A slowly closing pipegon



radius of the pipe in which it "rolls." Is this possible? Can it be done? I asked PIPEGON to do it, and the results are shown on the next page. What is happening? Are these pictures of real or imaginary machines? How can you work out your answer? Can you sketch it?

Recapitulation

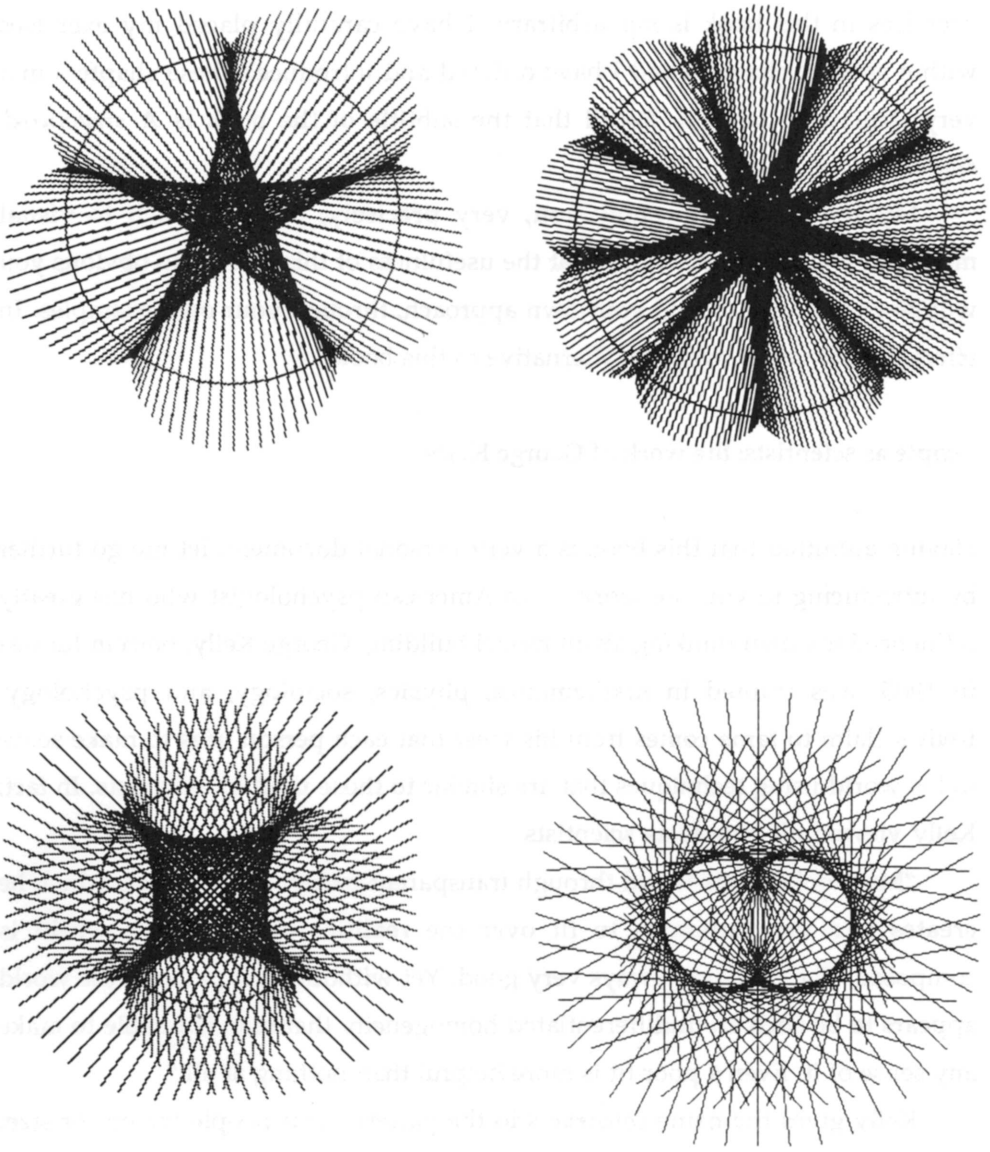
Let's summarize what has happened in the last few pages. We have built a Logo model that can produce a large variety of images, some of them very surprising. But more important, we have seen how the act of modeling can facilitate the visual exploration of some of the characteristics of a real-world machine. Once we began to tinker with our model, we wanted to tinker further. Some of our designs posed difficult questions whose answers were not at all obvious. Closure was such a question. We needed to do more tinkering and more experimenting to come to grips with what was going on.

Could we have predicted the directions this tinkering and experimentation would take before we started? I don't think so. Once we begin to model parts of our world, the act of modeling takes on a life of its own. I think we have touched what Lévi-Strauss said happens when one plays with miniatures. Model play "gratifies the intelligence and gives rise to a sense of pleasure which can already be called aesthetic." I hope you would also describe visual modeling as fun.

Why pipes and rollers?

I started with this particular machine because I was interested in it. I could have used any number of alternative illustrations, but this was my own direction. I will show you, in the chapters to come, dozens of other examples that illustrate the ways in which visual modeling encourages the modeler to *look* at the world differently.

Imaginary PIPEGONs?



But please keep the following in mind. All of the exercises are based on my own interests and the interests of my students. However, the ordering of these exercises in this book is *not* arbitrary. I have carefully placed my exercises within specific chapters and I have ordered and introduced these chapters in a very structured fashion. Recall that the subtitle of the book is *A structured approach to seeing.*

My intention is to show you, very precisely, one approach to visual modeling and graphics. I hope that the usefulness of this will be so obvious you will be encouraged to find your own approach, through your own examples. In effect, I want you to write an alternative to this book.

People as scientists: the work of George Kelly

Having admitted that this book is a very personal document, let me go further by introducing to you the work of an American psychologist who has greatly influenced my own thinking about model building. George Kelly, born in Kansas in 1905 was trained in mathematics, physics, sociology, and psychology. Kelly's claim to fame comes from his view that each person tries to make sense of his world using techniques that are similar to those used by scientists. In fact, Kelly viewed all people as scientists.

"Man looks at his world through transparent patterns or templets which he creates and then attempts to fit over the realities of which the world is composed. The fit is not always very good. Yet without such patterns the world appears to be such an undifferentiated homogeneity that man is unable to make any sense of it. Even a poor fit is more helpful than nothing at all."

Kelly gives the name *constructs* to the patterns that people try on for size. He could just as well have called these patterns *models*. These constructs—or models—"are ways of construing the world. They are what enables man, and the lower animals too, to chart a course of behavior, explicitly formulated or implicitly acted out, verbally expressed or utterly inarticulate, consistent with

other courses of behavior or inconsistent with them, intellectually reasoned or vegetatively sensed."

Each person's scientist aspect encourages him to "improve his constructs by increasing his repertory, by altering them to provide better fits, and by subsuming them with subordinate constructs or systems." For Kelly, human *behavior* is the application of *scientific method* in making sense of a particular environment. Rather than merely responding to surroundings, people use an experimental approach to test and extend their system of personal constructs. Each person's goal, in Kelly's view, is to build explanatory models that effectively explain and predict personal environments.

Kelly suggested shortcuts for improving construct systems. Kelly's shortcut was to encourage individuals to make their own constructs verbally explicit. His most famous method for eliciting and verbalizing personal constructs is known as the repertory grid technique. Using slightly different words, Kelly's techniques encouraged individuals to build *verbal models* of their own constructs. Once built, these verbal models could be analyzed in much the same way as we have analyzed our pipe-and-roller model. Tinkering with constructs would occur naturally, and this would encourage further tinkering. And as a result of this play, construct models might become more general and more powerful.

Kelly worked with verbal rather than visual models, but many of his ideas can be extended to the latter. My interest, as was Kelly's, is to suggest how to describe our inner models. While Kelly was interested in the verbal description of models, I am interested in more graphical descriptions. My goal is to encourage you to look at your own visual baggage. Obviously, I need words, too, to help in my form of elicitation. Sometimes, you may think that I rely on words too much. Too much chat, you might say. . .

If you are intrigued by this very brief account of George Kelly's work, find his book *A Theory of Personality: the psychology of personal constructs* (W. W. Norton, New York, 1963). All the Kelly quotes were taken from it.

Exercises

Exercise 2.1

Can you come up with some rules about PIPEGON closure? Specifically, can you characterize a final pipegon image in terms of the dimensions of its parts? Experiment a bit. Try to make some generalizations. Do the generalizations hold up after more experimenting? Whether you feel successful in this activity or not, find the following book in your local library: E. H. Lockwood, *A Book of Curves* (Cambridge University Press, Cambridge, 1963).

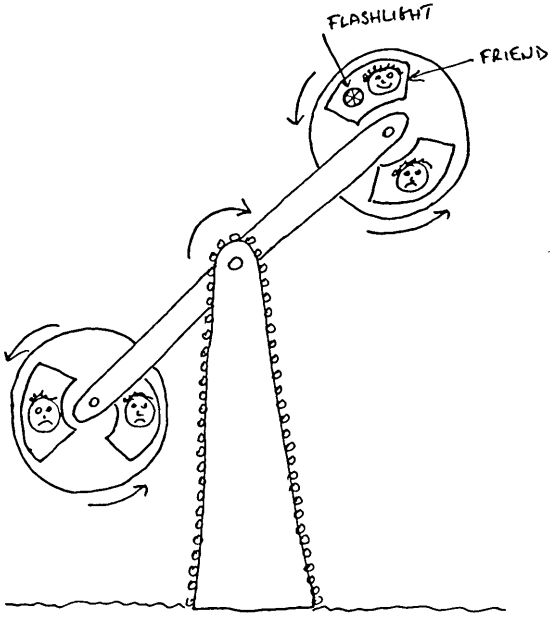
This book may help you think about closure. It may also suggest other image ideas to think about visually. Don't worry too much about the book's math. Look at the diagrams, and read the chapter names. Listen to these chapters: cardioids, limaçons, astroids, right strophoids, tractrices, roulettes, and glissettes. What images do these names dredge up? Sketch them before you find the book.

Exercise 2.2

How do you feel about carnivals and amusement parks? Do you enjoy their mechanical rides? I'm not talking about tame rides, like the merry-go-round or carousel, but wild rides that yank the rider through space. On the next page is a sketch of a machine that gave me a dose of healthy terror.

Suppose that we are watching this machine from a safe distance. A brave friend is sitting inside it and pointing a very bright flashlight at us. What pattern will this light trace out as the machine grinds into life?

The pulls and pushes on the rider of this machine change suddenly and unexpectedly. Can you make a picture of this? Can you describe visually why this kind of machine is so scary?



Exercise 2.3

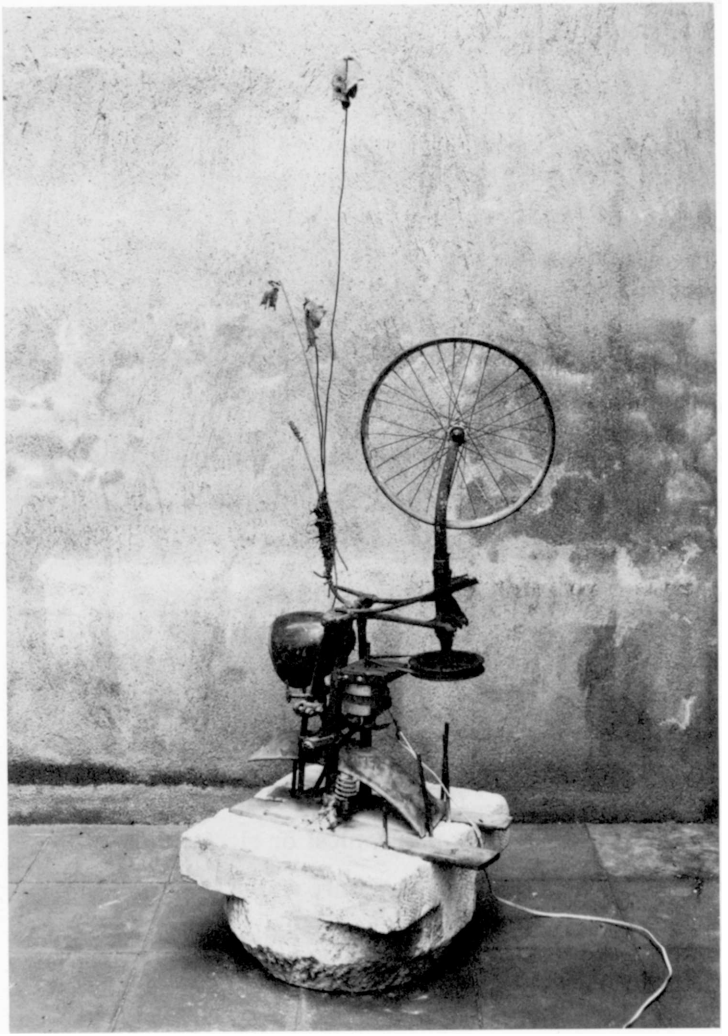
Design some *imaginary* carnival rides and give your machines imaginative names. Draw big sketches of your ideas; draw them large enough so that others can "read" them. Describe the ride in words so that potential travelers will know what to expect, *before* they climb aboard. You had better show them some pictures of the trip as well. Why not use Logo to generate these scenes?

Exercise 2.4

Do you know the term "kinetic sculpture"? If not, you can guess what they are, or rather what they do? Kinetic sculptures are mechanical or electronic sculpture-machines that move, clank, or flash. Some even squirt water (for example, the wonderful kinetic fountain designed by Nikki de Saint-Phalle and Jean Tinguely opposite the Pompidou Center in Paris).

Carnival rides are a special class of kinetic sculptures. They may not seem suitable for art gallery installation, but I have seen films of amusement park rides included in exhibitions. The French sculptor Jean Tinguely does kinetic sculpture on a more modest scale. Below is a reproduction of his "Homage à Marcel Duchamp," done in 1960. It is human scale, about 5 feet high.

Design and build a kinetic sculpture using Logo. You might start by trying to model the Tinguely machine. PIPEGONs are a kind of kinetic sculpture, too.



Chapter 3
Visual Discovery

"The discoverer is the [person] who sees in familiar objects what no one else has seen before."
Norwood Russell Hansen

"In order to think productively about the nature of a fact or problem, whether in the realm of physical objects or that of abstract theory, one needs a medium of thought in which the properties of the situation to be explored can be represented."
Rudolf Arnheim

The style of discovery

This chapter continues the exploration of visual models and how they can stretch your mind-and-eye's vision into areas unfamiliar to you. The style of presentation remains the exercise form. Exercises will encourage you to experiment dynamically with the simplest of shapes. I hope to convince you, through illustrations, that playing experimentally with familiar and plain objects can lead you into a space filled with objects that are not only unfamiliar but surprisingly different from what you expected.

Problem-solving style

This book is structured around two kinds of exercises. First, there are many exercises described in the text to illustrate what I consider to be good problem-solving style. You don't have to do it my way, but I want you to see one person's way. Second, the exercise section at the end of each chapter offers several open-ended problems that demand originality and flamboyance. These exercises generally are not illustrated in the text; I don't want to influence your creativity by showing you how anybody else did them.

Let's begin this chapter, therefore, by finishing up some of the more specific exercises at the end of Chapter 1. From now on, at least one assigned problem from the previous chapter will be discussed in detail at the start of each chapter As I have already mentioned, much of the learning of a craft is acquired by watching and copying someone who is used to doing it. Constructive copying can lessen your programming anxiety by providing a firm support for personal exploration. (I've said that before, but it is worth repeating.)

Centered polygons with CNGON

Let's begin with a final look at Exercise 1.1. This problem asked you to refashion NGON. The new procedure, CNGON, should draw an n-sided polygon around a central point. CNGON, named for centered NGONs, will take two arguments: first, the radius of the circle circumscribed on the polygon, and second, the number of polygonal edges. Here is the completed version with plenty of comment lines, each introduced by a semicolon:

```
TO CNGON :N :RAD
 PU FD :RAD
 ; Move out to the circumference of circumscribed circle.
 RT 180 - (90*(:N-2)/:N) PD
 ; Turn right to face along one of the edges of the polygon.
 NGON :N (2*:RAD*SIN 180/:N)
 ; Draw an NGON with edge expressed in terms of the radius.
```

```
 LT 180 - (90*(:N-2)/:N)
 ; Turn left in preparation for backing down to center of
 ; figure.
 PU BK :RAD PD
 ; Back down to the center.
END
```

State transparency

What on earth does that mean? Notice an interesting feature of CNGON. After CNGON has drawn a figure, the turtle is returned to its starting place. CNGON puts the turtle back to where it found it (in terms of its screen location and its heading). We can also refer to the turtle's position and heading as its *state.* Because CNGON leaves the turtle in the same state in which it found it, CNGON is said to be state transparent. NGON, by the way, is also state transparent. We will find this notion useful later in this chapter, so remember the technical term. It's a nice visual description, as well.

The distance traveled so far

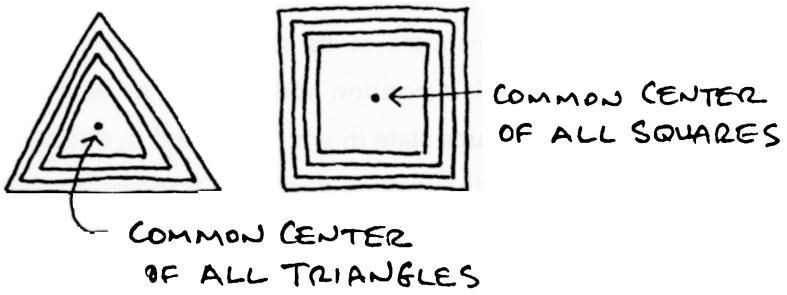
Recall, now, how you began your review of Logo mechanics. First, you drew boxes of fixed size. You then extended the idea of "box" to include polygon shapes of any number of sides and designed a procedure, NGON, to draw them.

CNGON uses the ideas of NGON but extends them. Our next step is to extend CNGON to make it more general. Let's think about generalizing CNGON to solve two of the exercises at the end of Chapter 1. Exercise 1.3 asked you to add the notion of *spin* to CNGON, and Exercise 1.4 asked you to fill up CNGON with other CNGONs. Let's start with the latter as it is a bit simpler and has one less argument.

Generalizing CNGON to draw concentric polygons

The diagrams accompanying the exercise suggested one way of filling up polygons: draw many increasingly smaller polygons, one inside another, until you have filled up the polygon with color or with texture.

The idea in sketches:



The idea in words:

Draw several polygons around a central point using CNGON. Allow each succeeding polygon to be a different size. If the radius of each succeeding polygon "shrinks" by 1 unit, the shape outlined by the largest polygon will gradually fill up with color. In summary, the new procedure will draw concentric polygons. The size of the first polygon will be set by one argument; the amount of size change of succeeding polygons will be determined by another argument; the number of sides to the polygons and the number of concentric figures will be determined by two additional arguments. Let's call this new procedure CONGON for concentric polygons. What will the first line of CONGON look like with all the needed arguments nicely arranged?

The idea as a Logo procedure:

```
TO CONGON :N :RAD :SLICE :TIMES
 ; CONGON stands for concentric NGONs.
 ; :N is the number of sides.
 ; :RAD is the radius of the first figure.
 ; :SLICE is the change in radius of succeeding NGONs.
 ; :TIMES is the number of polygons to draw.
```

Once design ideas are written down in sketches and words, with all the arguments defined and named, writing the Logo procedure becomes very easy.

CONGON needs a structure to execute CNGON as many times as is specified by the argument :TIMES. We can use recursion for this, similar to its use in SPINGON in the last chapter.

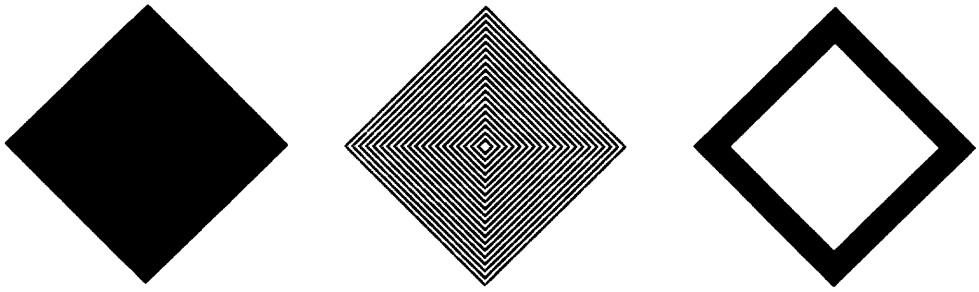
```
TO CONGON :N :RAD :SLICE :TIMES
 ; Concentric polygon exercise.
 IF :TIMES < 1 [STOP]
 CNGON :N :RAD
 CONGON :N (:RAD-:SLICE) :SLICE (:TIMES-1)
END
```

Experimenting with CONGON

First, fill up a square of radius equal to 80 totally with color. Second, fill up a square of radius equal to 80 with 20 concentric squares. Make each succeeding square 4 units smaller than the previously drawn one. Third, fill a square frame of radius equal to 80 and width equal to 20 with black. If you have a color Logo, you can set the pen to the color you wish. I am sticking with black and white.

Here are the three commands; the results are shown on the next page:

```
CONGON 4 80 1 80
CONGON 4 80 4 20
CONGON 4 80 1 20
```



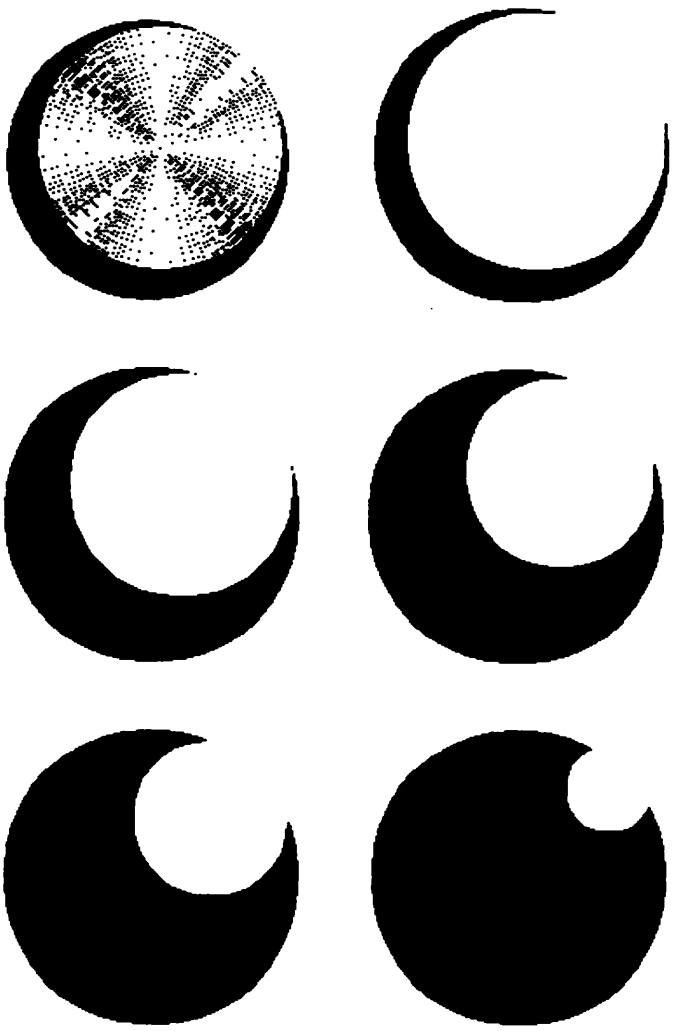
A crescent machine

A Turkish student of mine decided to make a screen collage of Turkish flags. The flag is red with a large star and crescent in the center.

"I decided to work first on the crescent shape. I knew I could do this by overlapping circles. This is, after all, how the crescent is formed in the sky: the circle of the earth's shadow falls across the circle of the moon's face. My plan was first to draw the moon with CONGON using some appropriate color, and then to draw a second smaller circle that partially overlaps the first. The second circle will use an erasing color. These double circles would simulate the shadow of the earth erasing part of the moon's image as it passes across it. I like how this little Logo machine would be simulating the sky machine in order to produce what I wanted.

"In order to play a little with the size and placement of the overlapping circle, I had to design an EXPLORE type of procedure. Here are a few of the results. The very first image is designed to show you the *textures* of the two circles: the visible moon circle, and the invisible shadow circle. This image will help you read the remaining images. On the other hand, it may remind you of a large radish with one end sliced off.

"After going through these experiments, I've decided against doing the entire flag. I'm going to concentrate on the lunar shapes."



Generalizing CONGON to draw concentric polygons that spin

Now for Exercise 1.3, spinning polygons inside other polygons. We could edit CONGON to include a turn command after each CNGON is drawn. We need one additional argument: the angle of turn between one polygon and the next. Call this :ANGLE. Here is the edited version with the changed items indicated:

```
TO SPIN.CONGON :N :RAD :SLICE :TIMES :ANGLE
 ; Concentric polygon exercise.
 ; Note new name and new argument :ANGLE.
 IF :TIMES < 1 [STOP]
 CNGON :N :RAD
 RT :ANGLE
 ; Single new command.
 SPIN.CONGON :N (:RAD-:SLICE) :SLICE (:TIMES-1) :ANGLE
END
```

Is this procedure state transparent? Well, no. The turtle always returns to the center of the figure, and that was where it started from, but there is no guarantee that the turtle will end facing in its original direction. The turtle turns a total amount, to the right, from its starting heading of `:TIMES*:ANGLE` degrees. To make this procedure state transparent, we need to turn the turtle back to the left by this amount after all the polygons have been drawn. Can we insert such a command in `SPIN.CONGON` above? The problem is that the `:TIMES` argument is decreased by 1 each time recursion is used in the procedure's last line. This is the stopping mechanism. When `SPIN.CONGON` is finished, the value of `:TIMES` is zero.

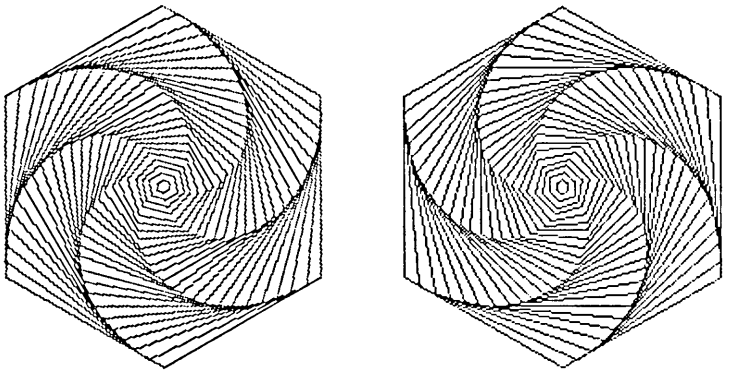
We could embed `SPIN.CONGON` in another new procedure, which we call `SPIN`. Why is `SPIN` able to use the original value of `:TIMES` when `SPIN.CONGON` isn't? Note: there are other ways of handling this state transparency problem. This is my way; I like it because it can be understood easily. It's also very tidy because it doesn't demand any more arguments or the introduction of local variables—which we haven't discussed yet.

```
TO SPIN :N :RAD :SLICE :TIMES :ANGLE
 ; State-transparent SPIN.CONGON runner.
 SPIN.CONGON :N :RAD :SLICE :TIMES :ANGLE
 LT :TIMES*:ANGLE
END
```

`SPIN` does only two things. It runs `SPIN.CONGON` and then returns the turtle's heading to its original state.

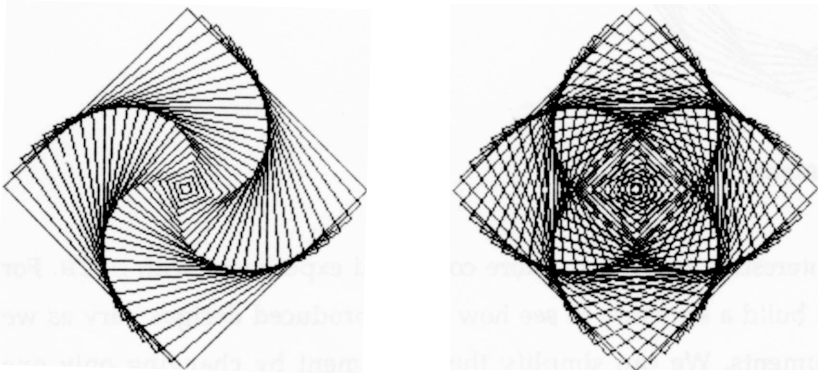
SPIN productions

The first figure was run with `SPIN 6 100 4 25 5` and the second with only the sign of the last argument value made negative. This changes the spin orientation.



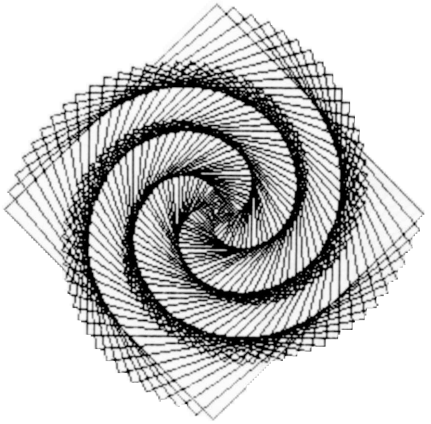
Squarish SPIN images

The left figure was produced with `SPIN 4 100 4 25 5`; how can we make it more complex? The right figure suggests a new kind of complexity that we haven't yet seen. It is a composite image that includes the left figure with its mirror image placed on top of it.

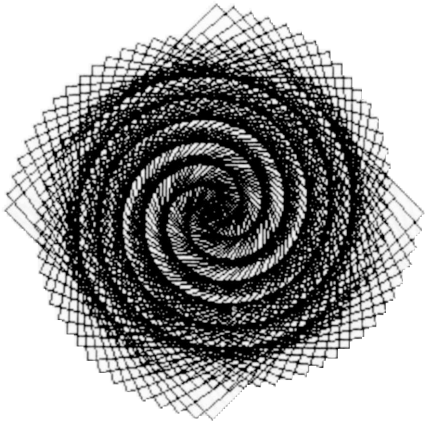


More squares

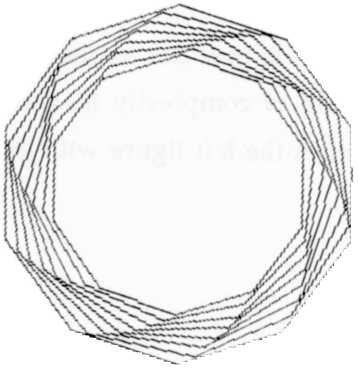
```
SPIN 4 120 2 60 5
```



```
SPIN 4 120 1 120 5
```



A winter holiday wreath



Visual experiments

It might be interesting to set up a more controlled experiment with SPIN. For example, let's build a structure to see how SPIN-produced designs vary as we alter the arguments. We can simplify the experiment by changing only one

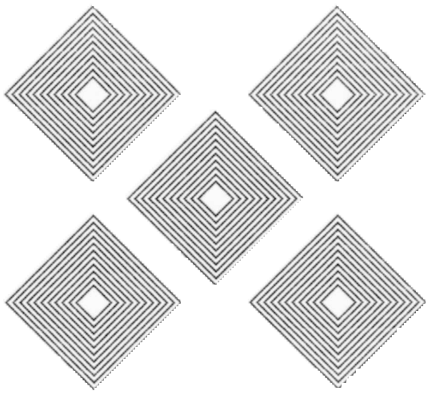
variable at a time I have decided to look first at the effect of an altered :ANGLE argument.

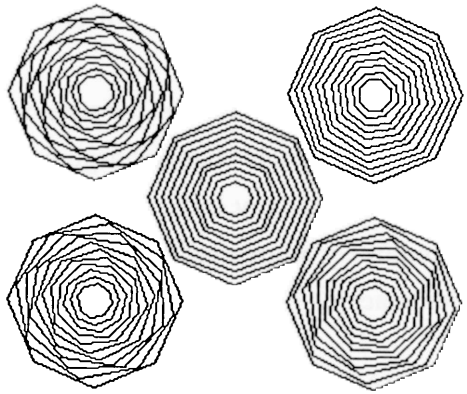
I thought it would be nice to see a number of SPIN.CONGONS on the screen at the same time, so that I can compare the effect of the single changed argument. By looking at the size of my screen, and deciding that my individual figures would be about 50, I decided where (the cartesian x-y addresses) on the screen I would put them. Here is my visual simulation machine.

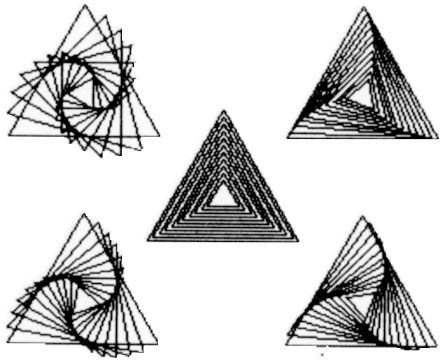
```
TO EXPLORE :N :SLICE :TIMES :A1 :A2 :A3 :A4 :A5
 ; To explore five different figures on the screen at once.
 ; Only the angle variable will be changed.
 PU SETXY  (0     0)    PD SPIN :N 50 :SLICE :TIMES :A1
 PU SETXY  (70   60)    PD SPIN :N 50 :SLICE :TIMES :A2
 PU SETXY  (70  -60)    PD SPIN :N 50 :SLICE :TIMES :A3
 PU SETXY  (-70 -60)    PD SPIN :N 50 :SLICE :TIMES :A4
 PU SETXY  (-70  60)    PD SPIN :N 50 :SLICE :TIMES :A5
END
```

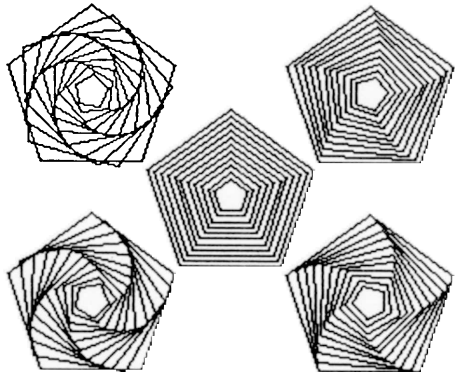
I don't much like using the cartesian system to move the turtle around the screen, but this seemed an easy and quick way to start experimenting. I'll do turtle-reference moving later.

Four experiments with EXPLORE









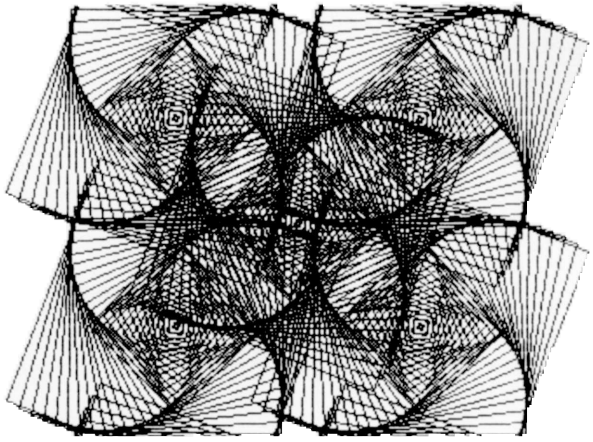
Some surprising images with EXPLORE

The figures on the last two pages looked isolated to me, so I decided to make them overlap by increasing the values used for the :TIMES argument. Having increased this value, I went back to my initial rule: keep all the arguments constant but one, the :ANGLE argument. Suddenly I had designs that were surprisingly intriguing. I think one reason for this surprise is that we can no longer predict the results of our experiment before we carry it out. And in addition, it is not at all obvious why the experiment proceeds the way it does.

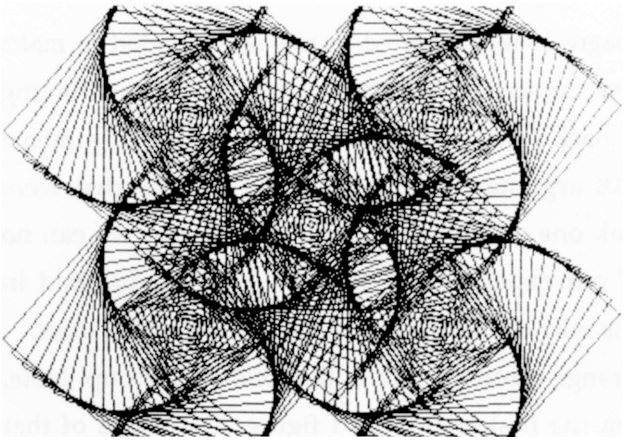
For example, note the strange central figure in the following group. Note, too, how it grows larger from the first to the third figure. The shape of that beast is very responsive to small changes in :ANGLE values. And the beast exists only within a narrow range of values. Our experimental machine has shown us something visually odd growing inside these designs. It has also indicated that this form of oddness is extremely sensitive to small changes in our experimental parameter. This is a sort of *visual sensitivity analysis.*

The argument values used are placed above each experimental result.

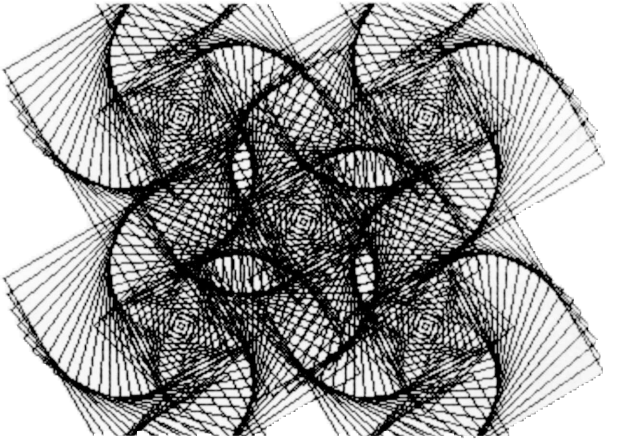
```
EXPLORE 4 4 40 4 4 4 4 4
```



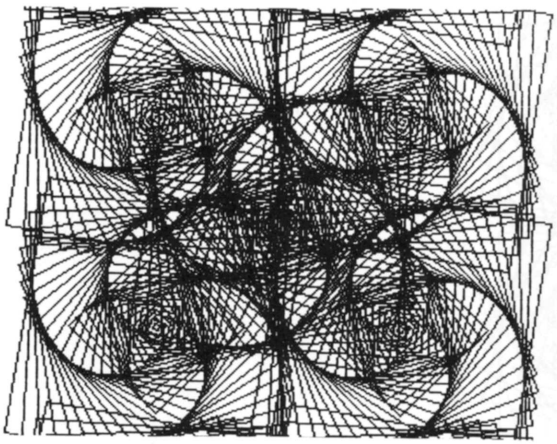
```
EXPLORE 4 4 40 4.5 4.5 4.5 4.5 4.5
```



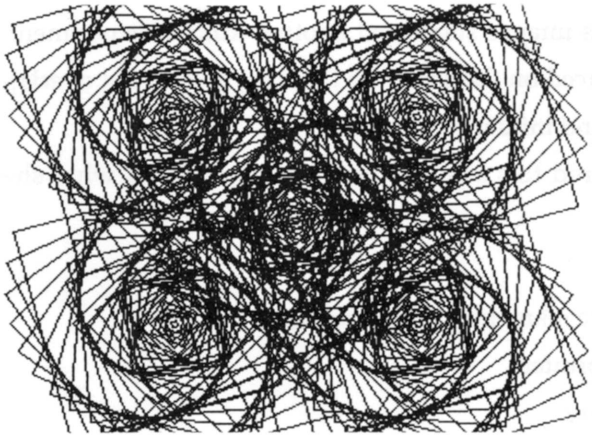
```
EXPLORE 4 4 40 5 5 5 5 5
```



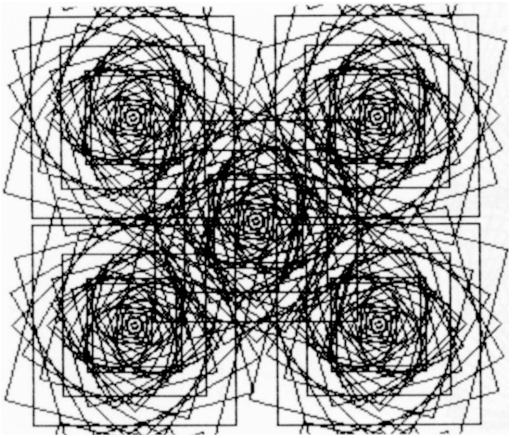
```
EXPLORE 4 4 40 6 6 6 6 6
```



```
EXPLORE 4 4 40 10 10 10 10 10
```



```
EXPLORE 4 4 35 15 15 15 15 15
```



Tiling spinning shapes

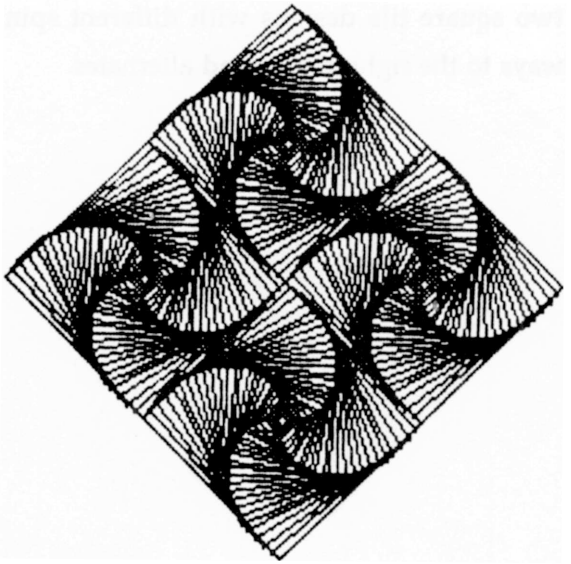
The overlapping of the previous images suggested another kind of experiment: could the spinning shapes be placed on the screen so that they would not overlap too much but would still touch and influence each other?

A student of mine considered this as a problem in *tiling*. Here's what she decided to do:

1. First, she used `SPIN` to produce many large single designs.
2. She printed these out on her printer.
3. She made dozens of photocopies.
4. She cut out the designs and placed them on a large piece of stiff paper.

She could then move the design cutouts around the paper, rotating and fitting them together to form larger, composite designs. This is similar to fitting decorated tiles together in various ways to create different patterns.

Here is the surprising result of one such cutout exploration. Four `SPIN.CONGON`s have been fitted together, like tiles, to form a larger design.



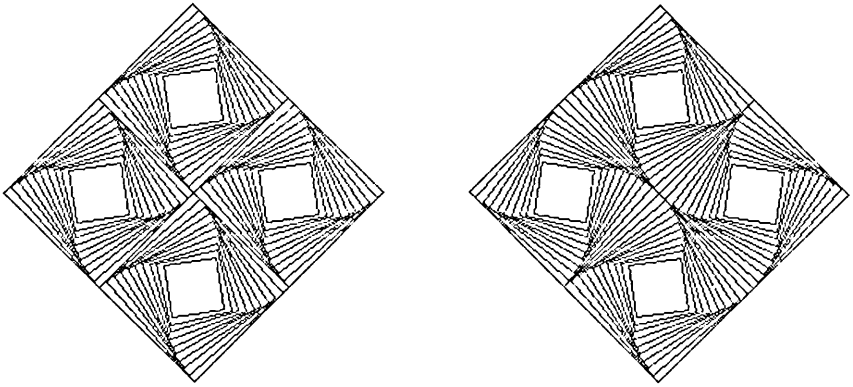
Logo-laid tiles

You probably detected that the above composite design could be done with a Logo procedure far more easily than with scissors and glue. I designed the following procedure to tile four square designs together. Any idea what that argument called `:ALT` is for? `:ALT` is short for alternating. If it is set to 1, all four of the tiles will spin to the right; that is, there will be no alternation of spin direction. However, if `:ALT` is set to -1, the odd tiles will spin to the left while the even tiles will spin to the right.

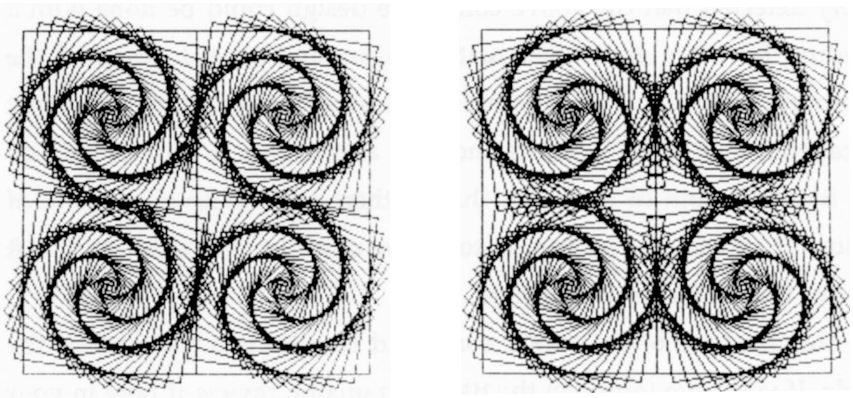
Notice the use here of the `MAKE` command to alternate the sign of the turning angle. If you have forgotten the `MAKE` command, review it now in your Logo manual.

```
TO LAY.SQ.TILES :RAD :SLICE :TIMES :ANGLE :ALT
 REPEAT 4 [PU FD :RAD PD -
           SPIN 4 :RAD :SLICE :TIMES :ANGLE
           PU BK :RAD -
           RT 90 -
           MAKE "ANGLE :ANGLE * :ALT]
END
```

The next illustration shows two square tile designs with different spin alternation schemes. The first is always to the right, the second alternates.



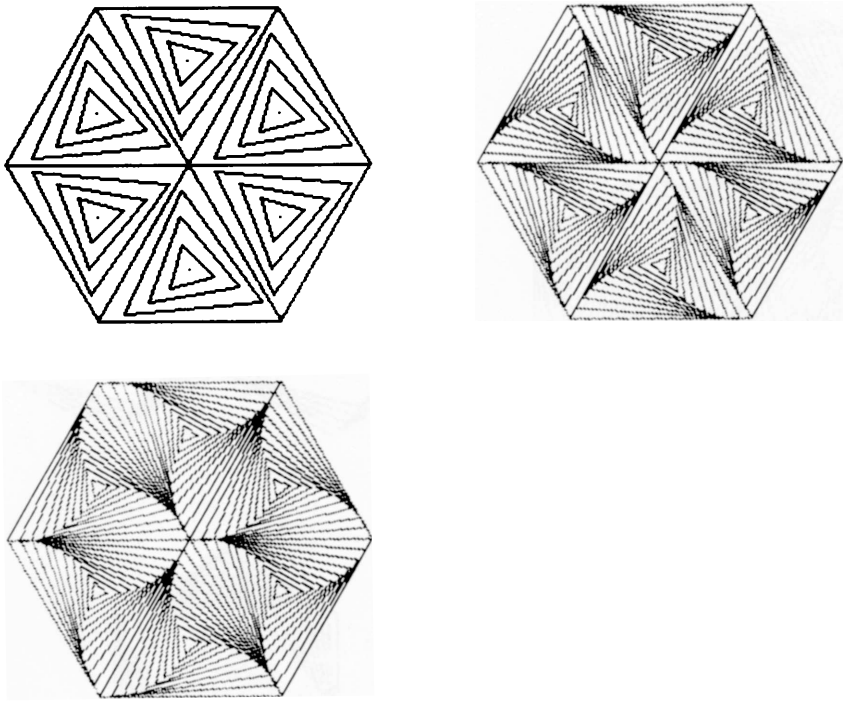
Here are two partially overlapping square tile designs. Note the appearance of more serpents.



Laying other tile shapes

Why not tile triangular shapes? The only quirk here is the necessity to turn the turtle 180 degrees to face the center of the composite design before drawing a `SPIN.CONGON`. Why is this turn necessary?

```
TO LAY.TR.TILES :RAD :SLICE :TIMES :ANGLE :ALT
 REPEAT 6 [PU FD :RAD PD -
           LT 180 -
           ; Orientation of turtle to face center.
           SPIN 3 :RAD :SLICE :TIMES :ANGLE
           RT 180 -
           ; Reorientation of turtle.
           PU BK :RAD -
           RT 60 -
           MAKE "ANGLE :ANGLE * :ALT]
END
```

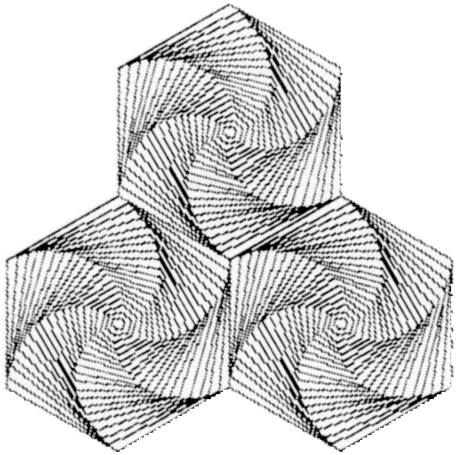


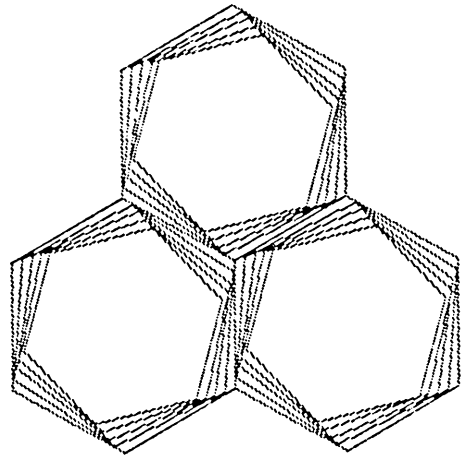
Hexagonal tiling patterns

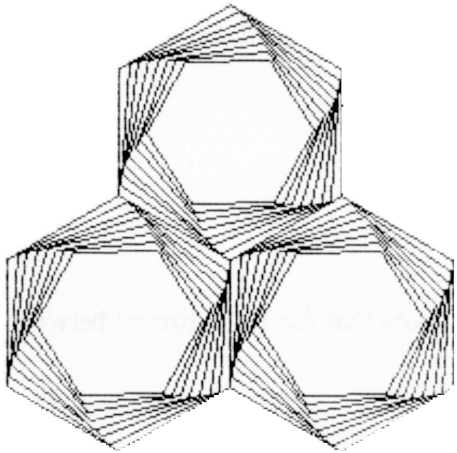
Why not? The procedure is straightforward. Note that the angle turned between tiles is now 120 degrees.

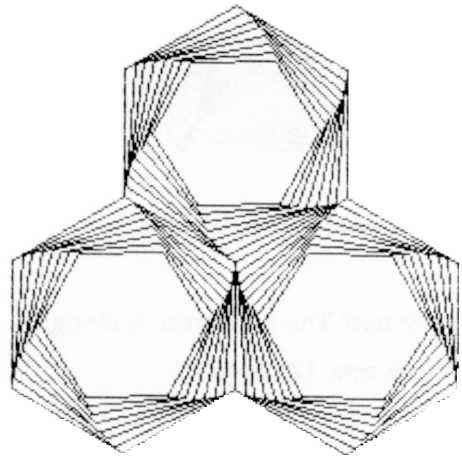
```
TO LAY.HEX.TILES :RAD :SLICE :TIMES :ANGLE :ALT
 REPEAT 3 [PU FD :RAD PD -
           SPIN 6 :RAD :SLICE :TIMES :ANGLE
           PU BK :RAD -
           RT 120 -
           MAKE "ANGLE :ANGLE * :ALT]
END
```

Something curious occurs now. Note that three of the four designs below look as if they are interlaced. Are they? What is happening?









Still only polygons

We have come a long way from the original BOX procedure of Chapter 1 to the surprising designs shown above. But do not forget that *all* the images we have created so far are still only polygons. Polygons, yes; but we have gone out of our way to play and experiment with these simple shapes, not just to complicate them but in the hope that we might see something new through them. Have you found the images of this chapter sufficiently strange to justify that hope?

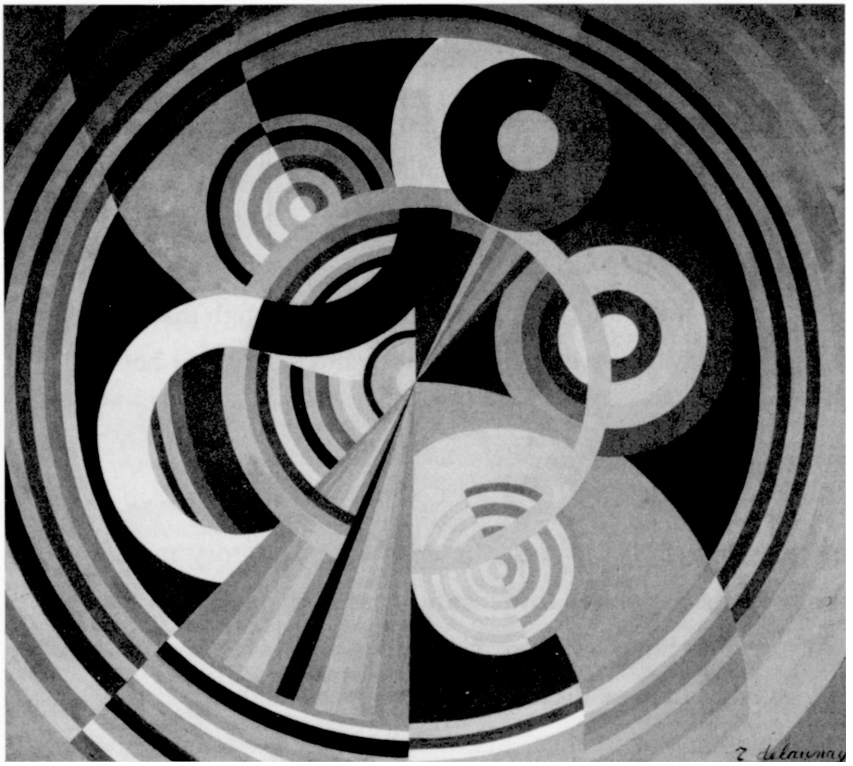
Extending the range of exploratory models

Before going on, I would like to describe another exploratory model. In the following case, a model is used to investigate some of the visual characteristics of a painting. So far, we have only looked at our own work. The results of the next exploration show that models can be effective visual thinking tools for analyzing somebody else's images. Specifically, this model will be investigating the qualities of balance in a design composition.

A Delaunay machine

Here is a reproduction of a 1938 painting, entitled "Disque," by the French artist Robert Delaunay. I recently discovered this work in a Paris museum. Although the thing is enormous, about 5 by 6 meters, I had never seen it before. Why, I wondered, did I like it so much?

I decided to think about this by building a model to "simulate" the picture. Obviously, I did not hope—or want—to reproduce the painting. I can always go and see the real painting whenever I want. Perhaps, I thought, I could fashion a model to explore some of the characteristics of a painting that I especially liked. And that might help me to see better what attracted my eye in the first place.



I would be exploring two things with my model: the world I was looking at in the Delaunay painting and the world of how I felt about it. I was sure that the latter would predominate. Why? Because I am convinced that models tell more about their builders than about what is being modeled.

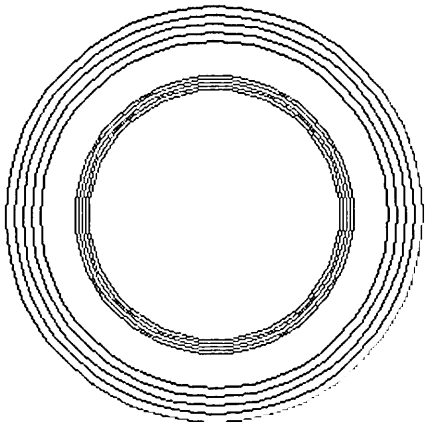
Painting elements

I decided against using too many design elements. The ones I selected were a bit arbitrary, but they were based on my impressions of the design elements used by Delaunay. Notice that none of my elements is an exact replica of *anything* in the painting. I built a different Logo procedure to draw each of these elements and to place them onto an imaginary canvas grid. Each element and its procedure is shown below.

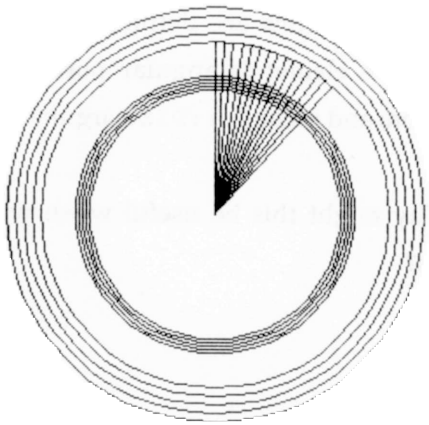
These procedures are simple and easy to read, but I want to add a few comments. I decided not to vary every possible dimension in my experiments. The double rings, for example, have a fixed size. However, while the form and size of the other design components are also fixed, each can be placed at different locations of the canvas. These design element locations are defined by two arguments: first, an `:ANGLE` argument sets the element's angular position, measured clockwise from the straight-up position; and second, a `:DIST` argument defines the elements distance from the center.

Each procedure is state transparent. Why might this be useful when we begin to use these procedures to paint canvases?

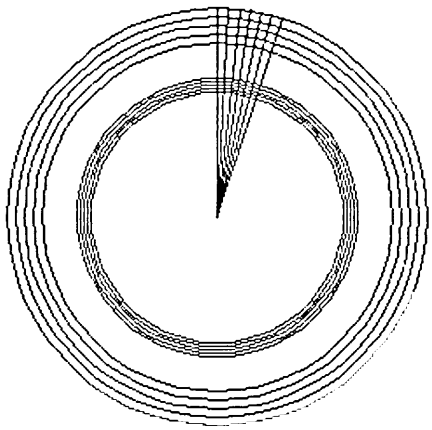
```
TO DOUBLE.RINGS
 CONGON 40 120 5 5
 CONGON 40 80 2 5
END
```



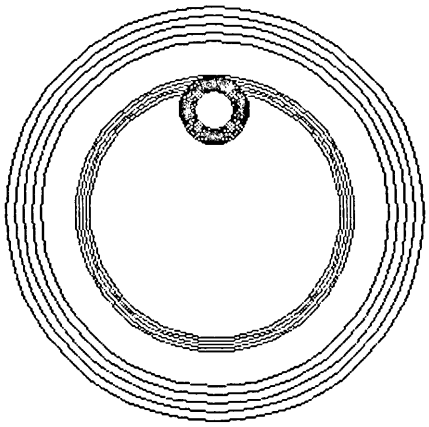
```
TO WIDE.SPLINES :DIST :ANGLE
 RT :ANGLE
 REPEAT 15 [FD :DIST BK :DIST RT 3]
 LT :ANGLE + 45
END
```



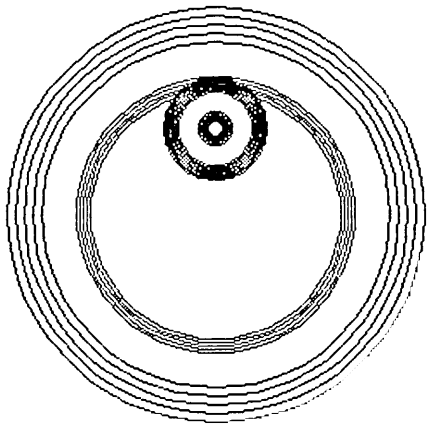
```
TO NARROW.SPLINES :DIST :ANGLE
 RT :ANGLE
 REPEAT 7 [FD :DIST BK :DIST RT 3]
 LT :ANGLE + 21
END
```



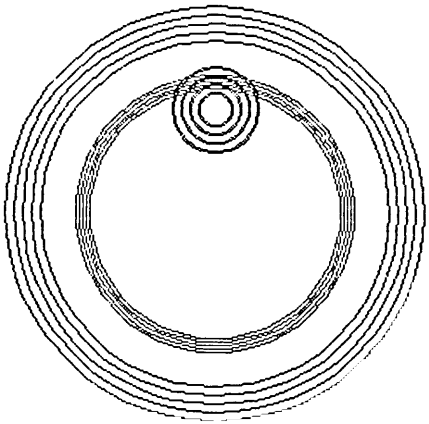
```
TO ONE.TARGET :DIST :ANGLE
 RT :ANGLE PU FD :DIST PD
 CONGON 40 20 1 10
 PU BK :DIST LT :ANGLE
END
```



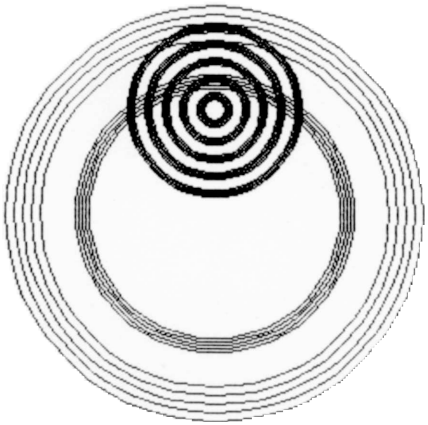
```
TO TWO.TARGET :DIST :ANGLE
 RT :ANGLE PU FD :DIST PD
 CONGON 40 30 1 10
 CONGON 40 10 1 7
 PU BK :DIST LT :ANGLE
END
```



```
TO FOUR.TARGET :DIST :ANGLE
 RT :ANGLE PU FD :DIST PD
 LOCAL "R MAKE "R 25
 REPEAT 4 [CONGON 40 :R 1 2 MAKE "R :R - 5]
 PU BK :DIST LT :ANGLE
END
```

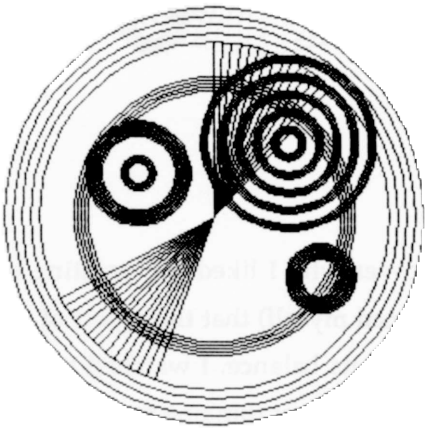


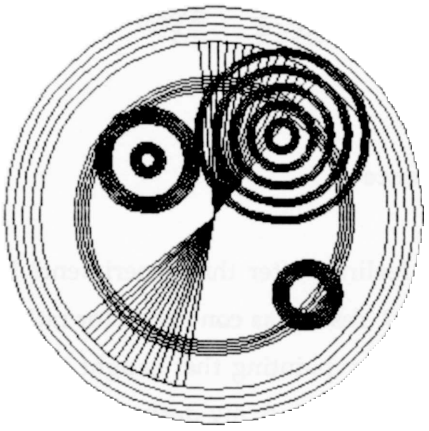
```
TO FIVE.TARGET :DIST :ANGLE
 RT :ANGLE PU FD :DIST PD
 LOCAL "R MAKE "R 50
 REPEAT 5 [CONGON 40 :R 1 5 MAKE "R :R - 10]
 PU BK :DIST LT :ANGLE
END
```

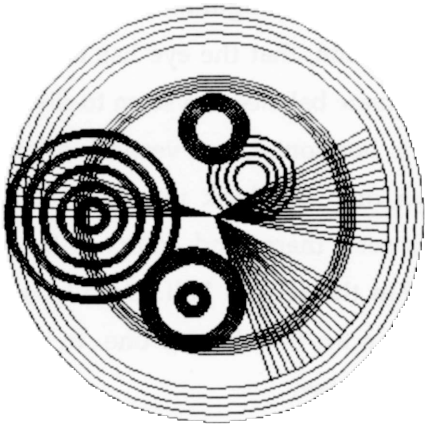


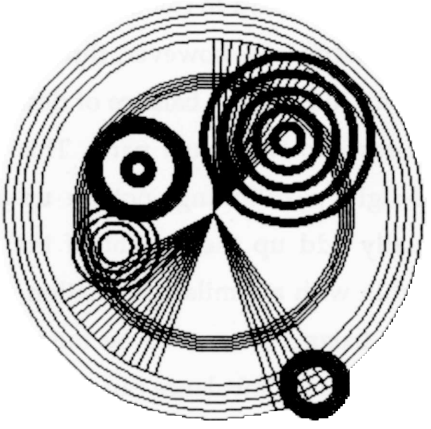
Delaunay experiments

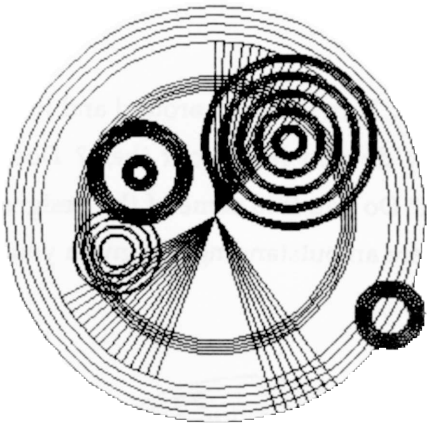
Six illustrations of Delaunay experiments follow. Turn the book around and look at the designs in different orientations. How do you feel about them? Does orientation of the page effect a specific design? Do you find some of the designs more pleasurable to look at than others? Is there an outstanding design, in your opinion? What makes it so?

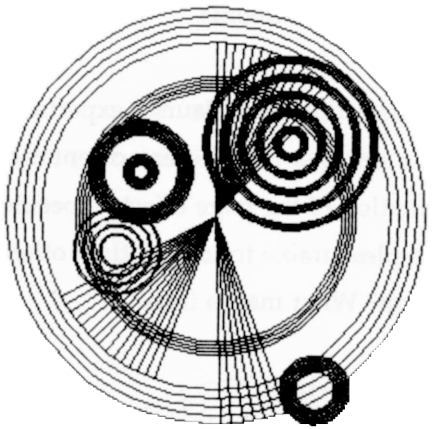












Balance

My feelings after this experiment? Did I discover why I liked the painting so much? Yes. I was convinced (or rather, I convinced myself) that the aspect of the Delaunay painting that caught my attention was its balance. I watched myself as I tried to place my design elements on the screen. I kept thinking to myself: try to balance the composition. This was difficult because the weight of each of my elements was hard to calculate until after I tried a new configuration.

I discovered, however, through experimentation, that the eye has its own rules for judging the balance of designs. To evaluate balance, we seem to divide pictures into two equal parts. This division is horizontal, not vertical; a left-and-right partitioning, not an up-and-down one. We look into each half, visually add up the weight of the elements seen there, and compare these weights with a similar calculation made within the second half. Usually, we find a balanced scene to be more pleasant than an unbalanced one. In fact, extreme visual imbalance can make me feel very uneasy.

How does the eye calculate the weight of a design element? Here are some rules. The relative size of shapes is important: bigger forms may be judged

heavier than smaller ones. But relative color value is also important. For example, white against black, because it attracts our attention, has more weight than gray against black, which attracts less attention. Thus a small black object might be heavier than a large gray one, if both were on a white background.

Variety of shape is also important in weight calculations. A more complex shape, because it attracts the eye, may be judged heavier than a boring shape. For example, a small asterisk may have more weight than a larger circle. Relative position is important, too. For example, a small shape in an extreme position can sometimes balance heavy shapes that are more centrally located.

When all the elements of a scene seem to radiate from a central point, a kind of balance known as radial balance occurs.

Because the eye seems to be concerned with horizontal rather than vertical balance, rotating a picture can destroy its balance. Why? Or, on the other hand, a rotation can fix an unbalanced picture.

Now go back and look at the Delaunay painting. I see a calming image made from disparate elements, finely balanced into a single design. Turn it on its side and what happens? Now go back and look at the Delaunay simulations. Do they balance? How would you change each of them to make them more balanced? Try some balancing exercises yourself with a design machine of your own invention.

Polygons placed on the vertices of other polygons

There is one remaining exercise to discuss from Chapter 1. Exercise 1.5 asked you to design a Logo procedure that will draw polygons centered on the vertices of other polygons. You need to have a good feeling for what this problem is really about before you rush off and try to solve it. As always, small drawings can help by making problems easier to visualize and to manipulate.

Even the simple sketches below should make the problem clearer. You will see the pattern required. Start by drawing one polygon. Now draw a series, or a

layer, of polygons on the vertices of the original polygon. Now draw polygons on the vertices of the polygons just drawn. The sizes of the polygons in one layer could change from those of the previous layer—getting larger or smaller—or all the polygons could remain the same size. We could change the shape of the polygons as well, from triangles to squares to pentagons. But let's work only with similar polygons in any one design. We can let their relative sizes change, though, from one layer to the next.

Recursion again

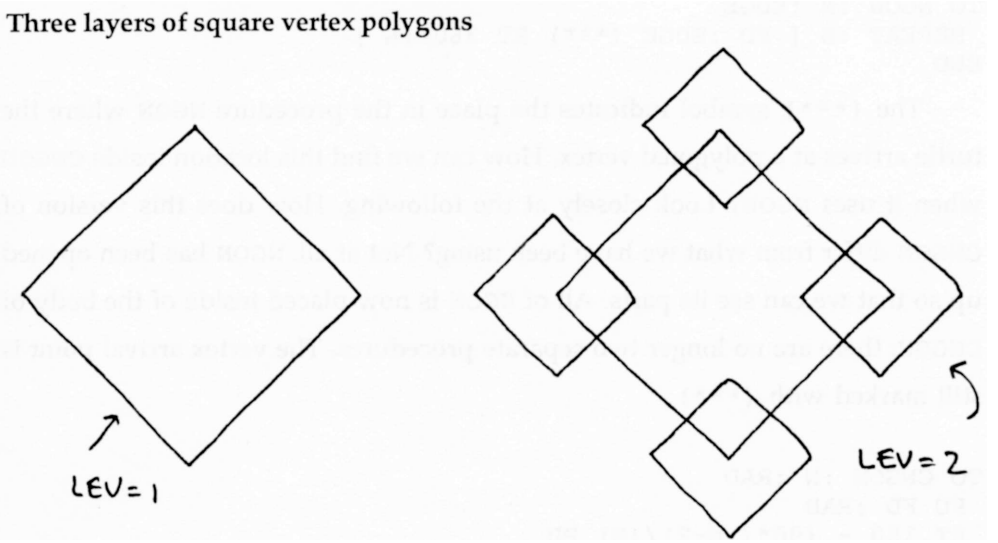
The hint given in Exercise 1.5 suggests that recursion, placed inside the body of CNGON, might be helpful. Let's look again at CNGON.

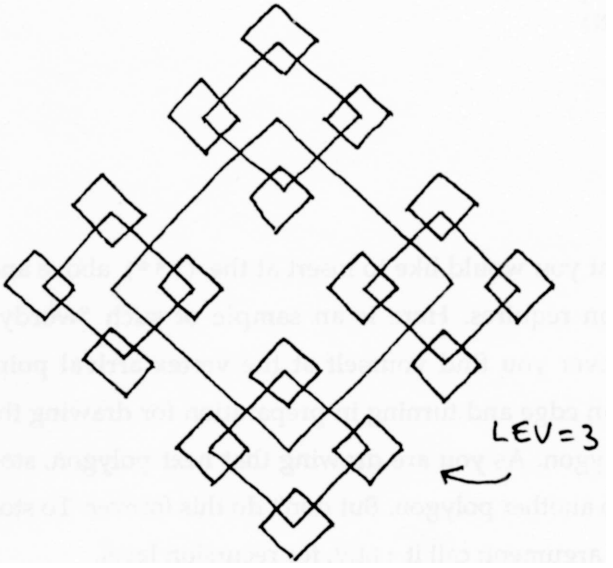
```
TO CNGON :N :RAD
 PU FD :RAD
 RT 180 - (90*(:N-2)/:N) PD
 NGON :N (2*:RAD*SIN 180/:N)
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
END
```

We want to draw centered polygons (CNGONs) around the vertices of other polygons. Let's find the place inside the CNGON procedure that corresponds to the turtle's arrival at a vertex. Then we can insert the command to draw another centered polygon around the turtle's current position on a vertex.

The vertex arrival point inside CNGON is the spot where recursion should take place: once a vertex is reached in the polygon drawing procedure CNGON, CNGON could be asked to draw another polygon around it; once this new polygon reaches a vertex, CNGON could be asked again to draw a polygon around the current position, and so forth. How can we find the vertex arrival point? NGON draws the polygon, so we will find the vertex arrival points inside NGON.

Three layers of square vertex polygons





```
TO NGON :N :EDGE
 REPEAT :N [ FD :EDGE (***) RT 360/:N ]
END
```

The (***) symbol indicates the place in the procedure NGON where the turtle arrives at a polygonal vertex. How can we find this location inside CNGON when it uses NGON? Look closely at the following. How does this version of CNGON differ from what we have been using? Not at all. NGON has been opened up so that we can see its parts. All of NGON is now placed inside of the body of CNGON; there are no longer two separate procedures. The vertex arrival point is still marked with (***).

```
TO CNGON :N :RAD
 PU FD :RAD
 RT 180 - (90*(:N-2)/:N) PD
 REPEAT :N [ FD (2*:RAD*SIN 180/:N)   (***)
             RT 360/:N ]
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
END
```

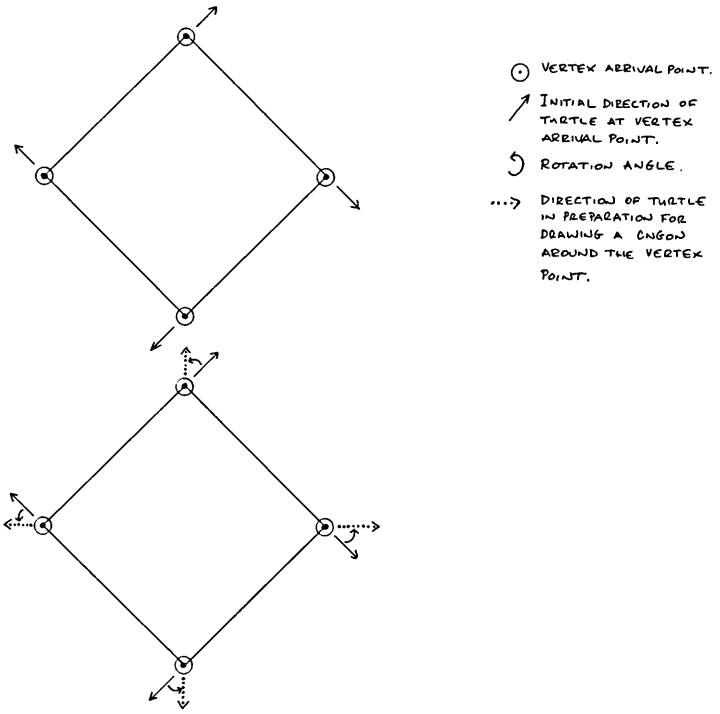
Vertex arrival point

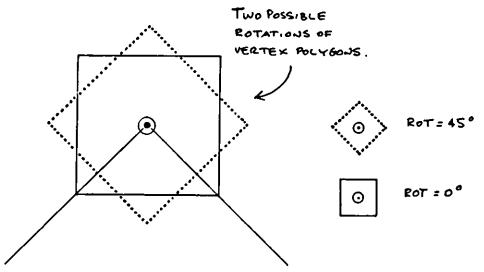
Now try saying in words what you would like to insert at the (***) above and what arguments the insertion requires. Here is an sample of such "wordy" thinking: "OK, turtle: whenever you find yourself at the vertex arrival point (the place between drawing an edge and turning in preparation for drawing the next edge), draw another polygon. As you are drawing that next polygon, stop after each edge and plan to do another polygon. But don't do this forever. To stop yourself, you will need a new argument; call it :LEV, for recursion level.

"You also will need to know if each succeeding layer of polygons is to get bigger or smaller in terms of of the preceding layer. So, define another argument; call it :FAC for growth factor. This will define for you whether to shrink or enlarge polygons from one layer to the next.

"Give the new procedure a new name, RECGON, for recursing polygons.

"One last idea. Maybe you should think about rotating the polygons at a vertex in relation to the previous polygon. Call this argument :ROT. Look at my sketches below."





The assembled RECGON

Now we need to convert CNGON into RECGON based on these words and sketches. Here it is, with plenty of comments.

```
TO RECGON :N :RAD :FAC :ROT :LEV
 ; Recursing polygons on vertices of polygons.
 IF :LEV < 1 [STOP]
 ; Stops the recursion at proper level.
 PU FD :RAD
 RT 180 - (90*(:N-2)/:N) PD
 REPEAT :N [ FD (2*:RAD*SIN 180/:N)  -
             LT :ROT -
             ; Orients turtle before drawing next figure.
             RECGON :N :FAC*:RAD :FAC :ROT :LEV-1 -
             ; Here is the inserted recursion apparatus.
             ; Note how the :RAD argument is scaled by :FAC.
             RT :ROT -
             ; Reorients turtle by removing rotation angle.
             ; Is state transparency an issue here?
             RT 360/:N]
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
END
```

Note that RECGON has five arguments. Therefore, each time RECGON is used, five values must be supplied to it. These values control the characteristics of the figures drawn. Only :RAD and :LEV change their values from one level of recursion to another. Can you see how these two arguments control the scaling of the RECGON design as well as its complexity (that is, level of recursion) ?

Recursion is a tricky business, but it is also very elegant. Many art students have told me that they find recursion aesthetically pleasing; they like its "shape," somehow.

Understanding recursion experimentally

Recursion is the most elusive concept in this book. I believe it is also one of the most powerful because it provides you with an entirely new metaphor for visual thinking. Working and thinking about recursion will encourage you to look-think at your world differently. In a strange way, recursive thinking gives you the power to *make* the world look different.

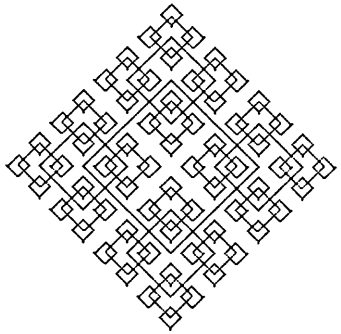
You may not understand recursion at all in the beginning, until you have experimented, over and over again, with recursive procedures written by others. If you can understand one recursive procedure well, you are on your way to using recursion on your own terms. Toward the goal of understanding a recursive procedure, you must design your own experimental apparatus to test it out.

Let's play a bit with RECGON. Watch how RECGON directs the screen turtle from one part of the design to another; the turtle is walking through recursion space. Try to get an intuitive feeling for how recursion "does it." At the end of this series of design experiments, we will talk more about the mechanical nature of RECGON and introduce you to a kind of visual model of recursion. For the moment, though, just play about with RECGON.

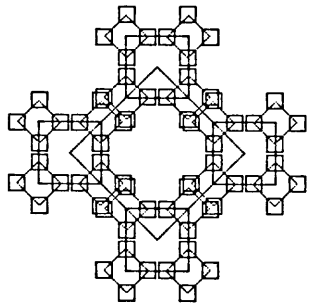
For example, ask RECGON to place squares at the vertices of other squares down to four levels of recursion. Four levels means that there will be squares on squares on squares on squares. The value that we will type for the argument :LEV will be 4. Now, what about scaling the design? If we make :FAC equal to .5, the squares will get smaller as recursion progresses. Finally, what about the rotation angle, :ROT? Let's try 45 degrees to start. To make this rotation idea less cryptic, we will experiment with different values to see what happens. One last item: set the beginning :RAD to 50 units.

Square RECGONs

```
RECGON 4 50 .5 45 4
```

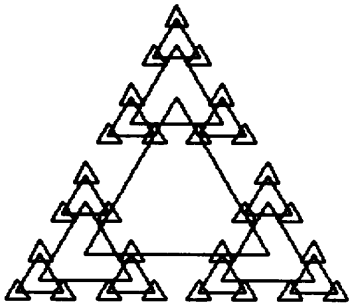


```
RECGON 4 50 .5 0 4
```

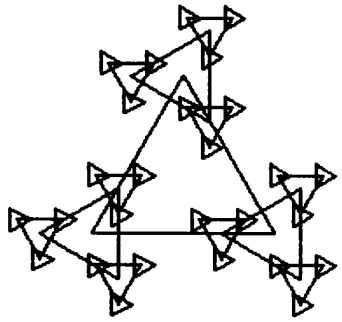


Triangle RECGONs

```
RECGON 3 50 .5 30 4
```



```
RECGON 3 50 .5 0 4
```



Less obvious RECGONs

Before going on to experimenting with funny values for arguments, to see what RECGON does, let me pose a question. Can you calculate the number of figures that RECGON must draw for any kind of polygon and any level of recursion? If you

could calculate the number of figures that must be drawn, you could guess how long any one design might take to draw. Let's try to attack this for a specific case. Later, you can generalize the approach.

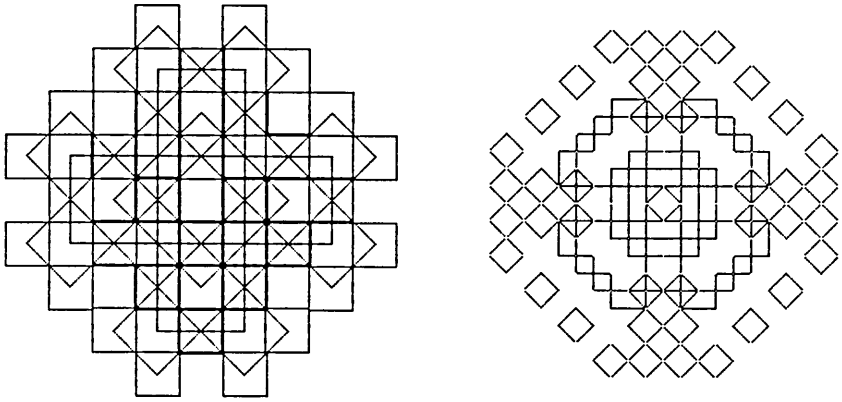
Think about square RECGONs. A level 1 RECGON would draw only one square, but a level 2 RECGON would draw an additional four squares on the vertices of the first: that makes five squares for level 2. Now, at level 3, we must put four squares on each of the squares drawn at level 2. That means another 4 times 4. In total, level 3 includes 16 + 4 + 1 = 21. Can you work out the general formula for number of figures in terms of n, the kind of polygon, and L, the level of recursion?

Here are two more RECGONs. The second design was drawn using the reversing pen color.

Can you verify that the second design was formed from 341 squares? (1 + 4 + 4*4 + 4*4*4 + 4*4*4*4 = 341.)

```
RECGON 4 50 (1/SQRT 2) 0 4
```

```
RECGON 4 10 (SQRT 2) 0 5
```



Recursion diagrams: visual models of recursion

We will be talking a lot more about recursion in the following chapters. But let us end this exploratory work with RECGON by presenting a visual model of how recursion works.

Before you read on, I need to show you my visual license. What follows is a *visual* rather than a technical description of recursion. What I describe as "happening" on the diagrams does not necessarily coincide with what is "happening" inside your computer. However, I have found that graphical recursion models are often far more useful in encouraging an intuitive recursion sense than technical ones. This is almost always true for art and liberal arts students, though my results are mixed with science students. But if you hanker after technical truth, take a computer scientist to dinner.

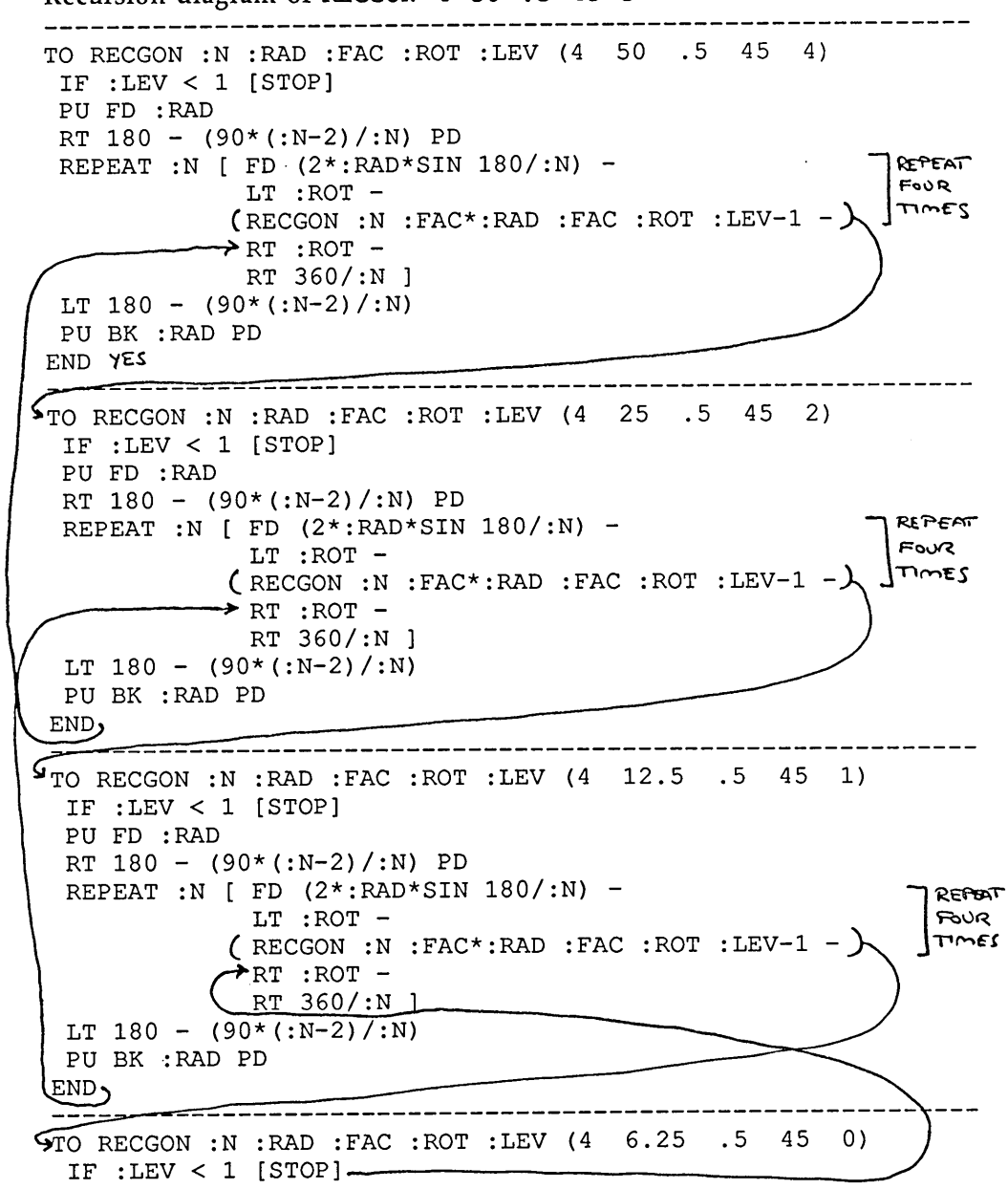
It is a little difficult to visualize how a procedure can be defined in terms of itself, so let's try the following. When a procedure asks to be run again, imagine that Logo makes a copy of the original procedure and locates this copy below the first. This copy is where the turtle goes in search of its next instructions. If the copy asks to be run again, a third copy is created and located under the second. The third copy is where the turtle goes in search of further instructions.

If the third copy does not call itself, the recursion goes no deeper and no more copies of the original procedure are made. The turtle climbs up from the third copy to the second copy and continues on the second copy looking for instructions. If the second copy does not call itself again, the turtle climbs up to where it left the first copy.

Think of these copies as sheets of paper on which the identical procedure is written. The turtle reads instructions from whatever sheet it is currently on. Suppose the turtle is reading instructions from the top sheet. Suddenly the top sheet calls itself. The turtle drops from the top sheet down to the sheet just below it and begins to read the instructions there. If this procedure ends without recursion, the turtle climbs back to the place on the top sheet where it had left off. The turtle, of course, carries the values of arguments along with it.

Here is a recursion diagram that sketches the path of operation after we type `RECGON 4 50 45 3`. Notice that the diagram shows two things. First, arrows indicate the path the turtle takes from one level to another; and, second, the values are indicated for the 5 arguments at each level.

Recursion diagram of RECGON 4 50 .5 45 3



Visual thinking

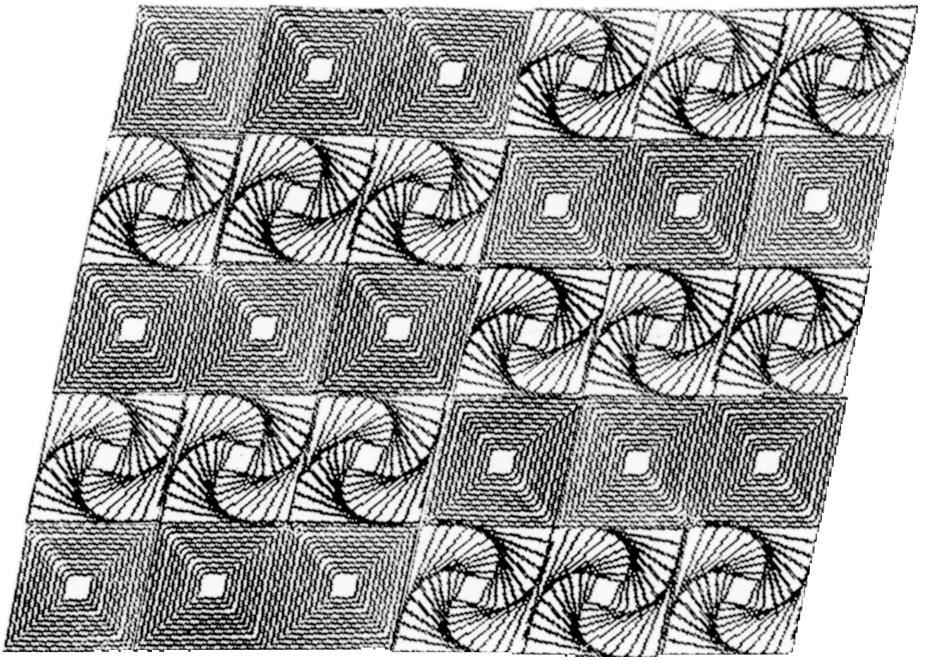
Do I need to summarize what we have been doing? I don't think so. The illustrations show it: a kind of visual thinking about objects and shapes.

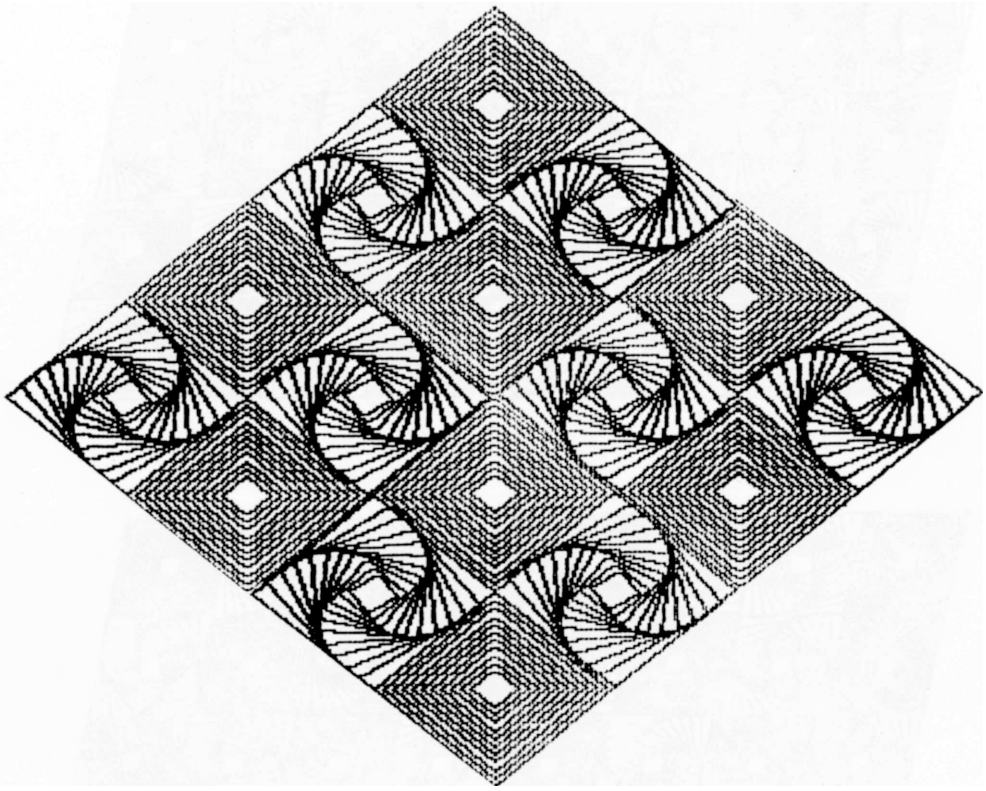
Exercises

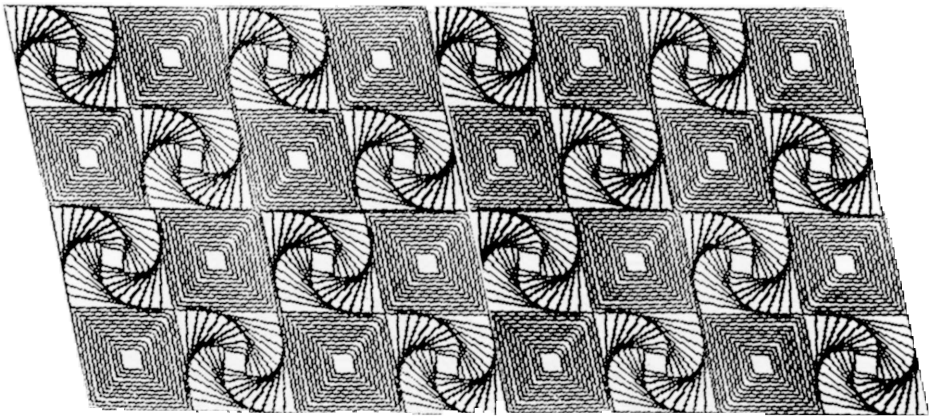
Exercise 3.1

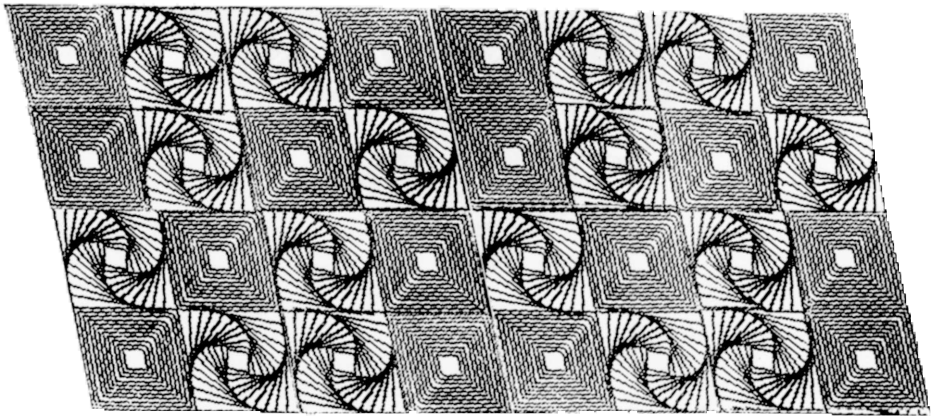
Look back over the composite designs that were made from the `SPIN.CONGONs` in this chapter. Design a Logo procedure that will produce composite designs from a number of different design elements.

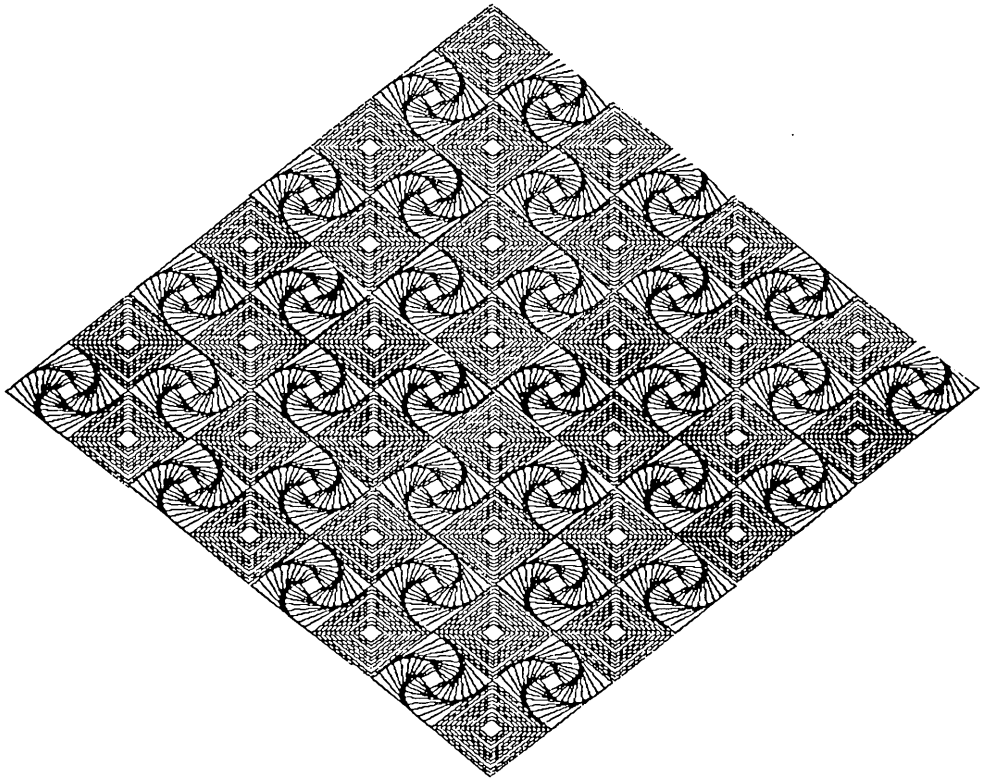
The following series of designs was done by a student. She labeled the collection "Hurricanes." They should give you some ideas. But don't limit yourself to figures based on the square.

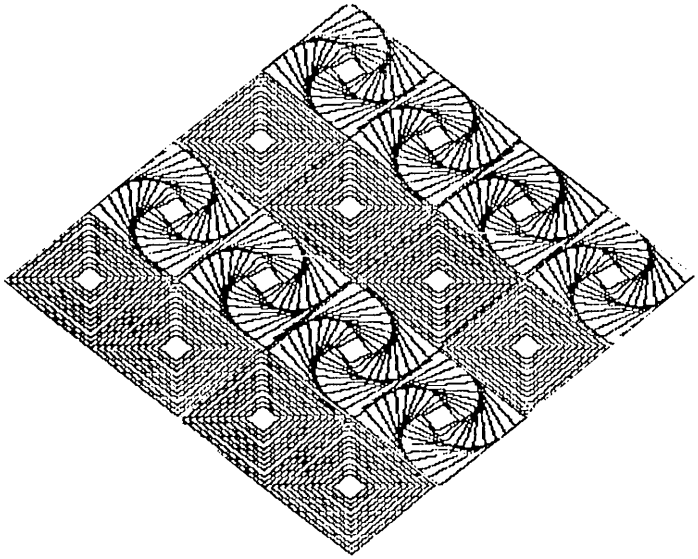




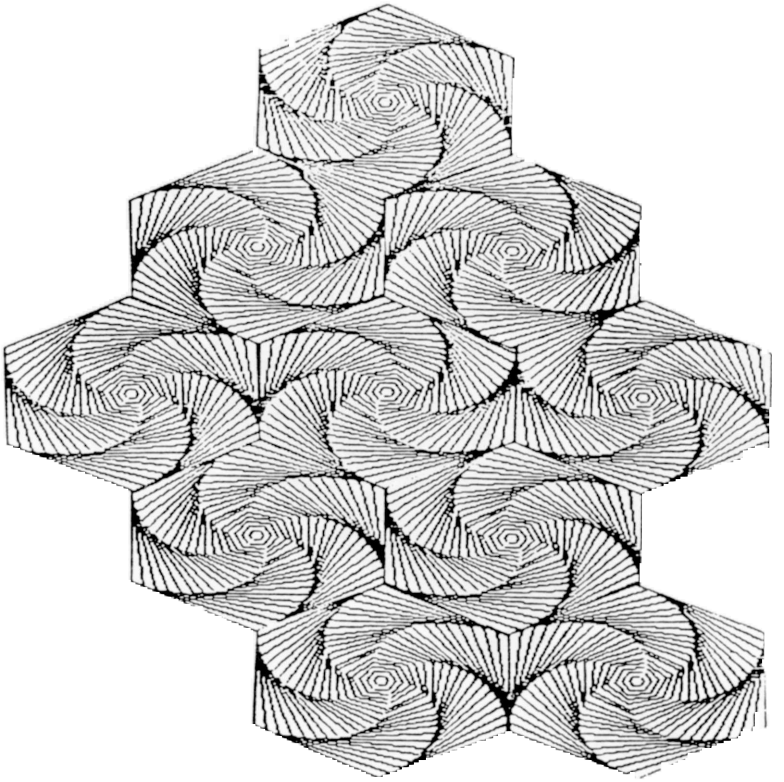








Here is another student "solution" to this exercise. He has constructed a map of France with alternatively spinning hexagons.



Exercise 3.2

Design a procedure that combines the ideas of SPIN.CONGONs with the ideas of RECGONs.

Exercise 3.3

Look back at the Delaunay machine. Invent a queasy machine that produces the most unbalanced designs imaginable. Try to characterize in words what your designs are doing and what you were doing.

Exercise 3.4

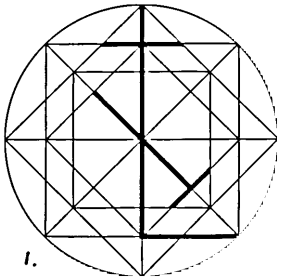
Build an exploratory model of a painting that interests you. Make sure you have a reproduction of the painting to carry around with you. Postcard images are the best. Go out to your local museum and buy a collection of reproductions by different artists. Don't limit yourself to geometric painters; include realistic work, even photographs. Select as large a variety of work as you can. Inspiration may strike from comparing two quite different pictures. Don't start doing any Logo until your painting collection is pinned up over your desk. Spend a lot of time looking at those postcards.

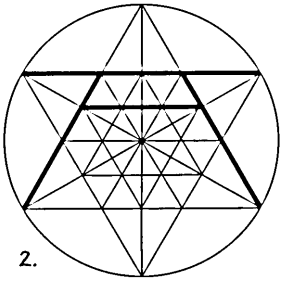
After you have finished this exercise, describe clearly what characteristics you have tried to simulate. Remember that you can simulate the characteristics of a painting without copying the image. Have your feelings changed toward your painting?

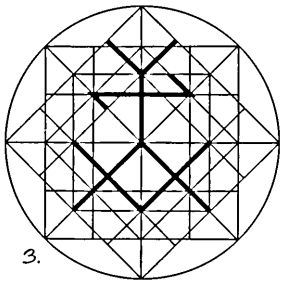
Exercise 3.5

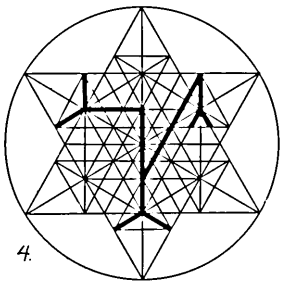
Study the following several pages of figures. Bring out your visual thinking gear; can you decompose these composites into their components?

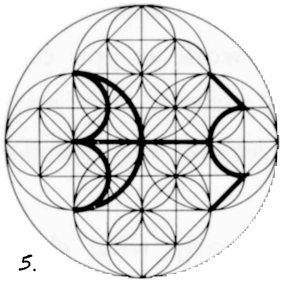
Stone mason marks

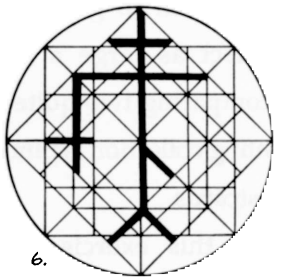


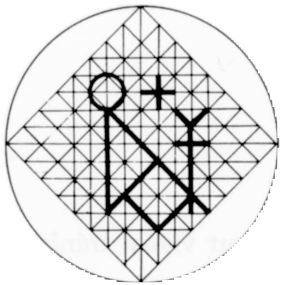


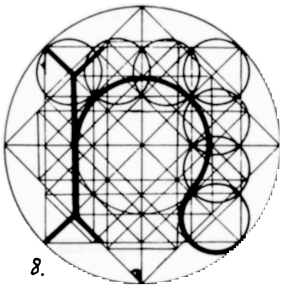




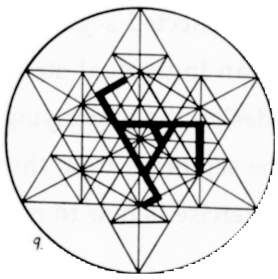


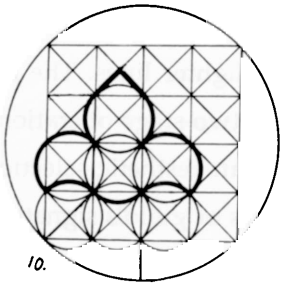


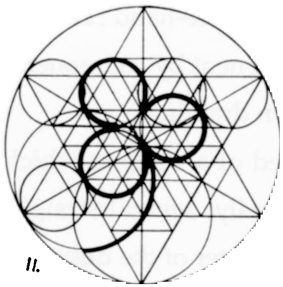


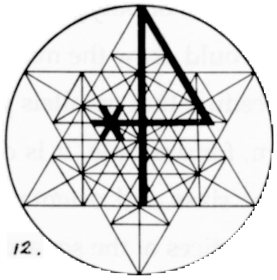


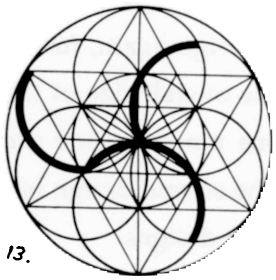
More stone mason marks

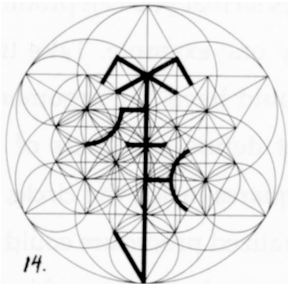


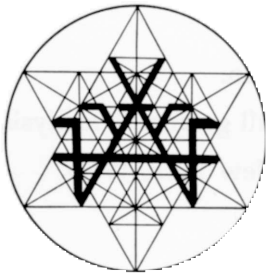


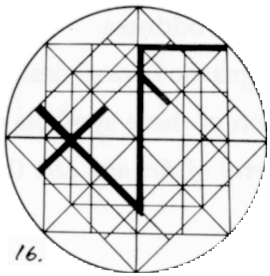












These symbols, called *stone mason marks*, are copies of the marks builders of Gothic cathedrals left as signatures on their work. The mark actually put on the stone is that indicated by the dark lines. These lines are selected segments from an underlying matrix of lighter lines. The design of an individual stone mason's mark was, therefore, a two-stage operation: first, design the underlying line matrix; and second, make an aesthetic selection of line segments from this matrix to express an individual's or guild's signature. The exercise here is to create the line matrices similar to the examples drawn.

Select one of the examples that strikes your fancy. Look hard at it. Think in terms of polygons and the placement of polygons. Make some free-hand sketches in your notebook on how you would draw the matrix, component by component. Next, try to design a Logo procedure that parallels your own drawing.

Take the very first design, for example. It is composed of a circle in which are placed squares of different sizes and orientations. Finally, several straight lines are drawn to connect the vertices of the squares to the center of the design.

You can do far more than just reproduce the designs offered here. Generalize your `MASON.MARK` procedures so that you can produce a suite of designs based on the theme suggested by any one example. Take the first design again, for an example. Your procedure could have one argument for the overall size of the figure, while another might define the kinds of polygons placed inside the circle. The first design has squares inside the circle, while the second design has triangles. Perhaps your generalized procedure could draw both designs.

Finally, you might want to design some kind of experimental apparatus that will display several of your designs on the screen at the same time, each with a slightly different degree of oddness.

If you find all of these designs too boring, design your own.

For a historical description of these symbols and a full geometric analysis see Matila Ghyka's *The Geometry of Art and Life* (Dover, New York, 1977).

Exercise 3.6

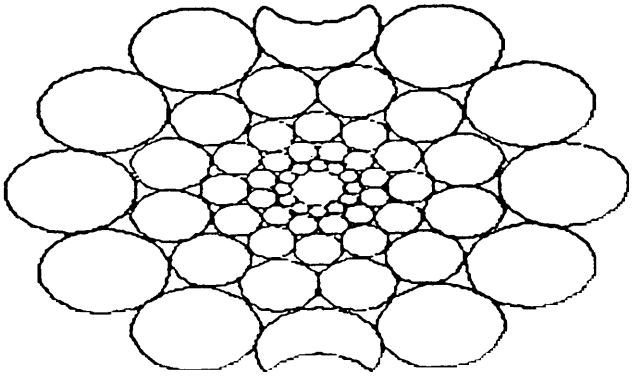
Design a personal mark in the manner of, but not limited to, the approach used by the Gothic stone masons. This mark should be a visual model of several aspects of your personality. This exercise will require a combination of introspection and Logo designing. You may find that your Logo designs actually help the looking within yourself.

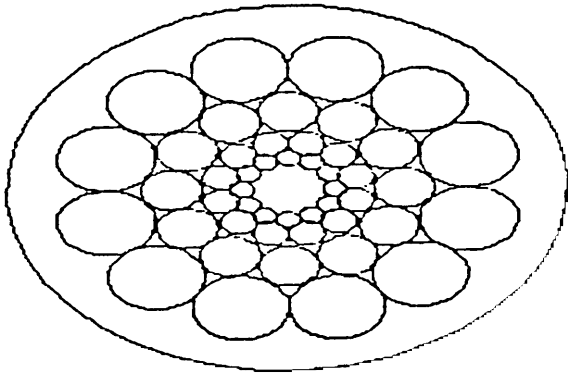
Start this exercise by doodling in your notebook before you write any procedures. Without thinking too much about what you are doing, write down a list of your own personality quirks. Do this as fast as possible. Now go back over your list and make small sketches of whatever images come into your mind. Try to make a sketch for each word. But if one word doesn't cause an image to pop into your mind quickly, leave it and go on to the next. Finally, combine some of your best sketches into a single design theme and design a Logo procedure to explore it. Don't intellectualize what you are doing. Do everything as quickly as possible. Think about what you have done only when you have generated some personality images.

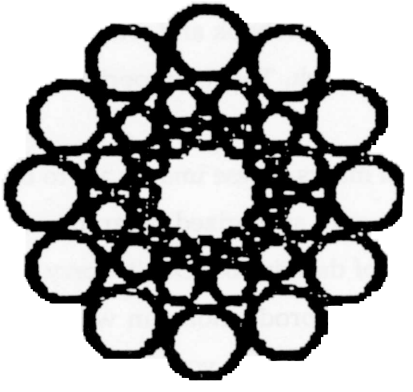
You may find that you will produce many personal marks that you aren't happy with. That's OK. Put them all in your notebook and write a short note on what you were trying to accomplish with each. This is important: keep a trace of what you are thinking and doing.

Finally, select one or two individual marks. These images are to be included in your published biography. This is the only authorized biography of your life and you may wish to write a few words of description. Don't worry about being too personal in your description: the book's production run will be very, very small. And since few people will read your comments, you can be totally candid.

The next page shows an example of a mark designed by one of my students. He found a basic theme that he presents in various configurations. Following the images is his description of what he was trying to say with the mark.







The student's personal mark explained in words:

"I liked the idea of using circles in my personal mark. I wanted to show that I have many interests and that no one single interest is most important to me. I try to be a well-rounded person (sorry about that). I get a lot of personal satisfaction from seeing how a number of different disciplines interact with each other. Math and art are two examples of my interests.

"I wanted the circles not to overlap in the design—I found that Gothic stone mason marks get too busy when too many lines overlap; I try to get my life pretty organized, so I wanted to show the circles in my head all touching—but in a nice neat way. I like being clever with math, so I was pleased to be able to work out the math in this design. I wanted the balls to get bigger and to fit nicely together. They do look a bit like ball bearings, I'm afraid. Maybe I come across to people as a bit of a machine. I don't know.

"I like the top mark because the turtle overlapped on the screen turning two of the circles into little moon shapes. These moons have rather soft outlines, and they would never work as ball bearings. The moons gave me the idea of softening up the entire mark—making it more organic and less geometric. I figured out how to have my printer make the first two designs look squashed; I liked the comparison with the nonsquashed third design. The bottom mark is done with CONGON. It looks like Greek jewelry that is made from those washer-like things.

"These three marks illustrate how I try to apply my mathematical mind to soft, organic (human) problems. And Greek jewelry? I think it is an example of math-inspired designs that serve humanistic ends."

The following is not an example of a personal mark!

Why? Because a physical likeness of the author is not the same as a model of the author's personality.

Chapter 4
Circular Grids

"Language is an instrument of human reason and not merely a medium for the expression of thought . . . ."
George Boole

"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems . . . ."
Alfred North Whitehead

Modeling for communication and thought

Like algebra, geometry, or any other language, the language of modeling is both a medium for communication and a notation system that helps structure thought. You can crystallize an idea about a shape or complex of shapes into Logo procedures. Logo gives you a language both to name the shape and to define how to go about drawing the shape. Once the shape is named in a procedure, that name can be used as a shorthand for the method of constructing and exploring the visual implications of the shape's definition.

In the preceding chapters you have seen that our visual intuitive sense is limited in many cases. That explains why some of the images are so surprising. Most people's visual vocabulary is fairly rudimentary, even when it comes to something as commonplace as polygons. Visual exploration with models can

help increase your visual vocabulary by forcing it to describe unexpected shapes. Have you felt visually stretched?

Unpacking and repacking shape packages

The goal of this chapter is to encourage you to take complicated designs apart in order to find their basic elements, the building blocks. Once unpacked, a complex design can be explored by modeling each of the component parts in turn. And once the small-part models are completed and named, you are free to concentrate on how to put the parts back together again, that is, how to repack the package with which we started. Along the way, you may be able to find several alternative ways to repack the pieces.

Do you remember the first time you took a small machine apart? I remember happily attacking a large, mechanical alarm clock. The pleasure came from solving two puzzles: to separate the pieces and then to put them back together again. And because I had seen through those puzzles, and had arranged all the small parts carefully on a carpet, I knew that the reassembled clock was not the same as the original.

Circular grids

Having explored polygons extensively, are you beginning to see the world packed with them? Were you able to spot and recreate in procedure forms the polygons hidden inside the stone mason marks of Exercise 3.5? I want to use this mason mark exercise as an example of how to go about the unpacking and repacking of complicated, composite images. The mason marks are good examples, too, of a special kind of composite shape organization. The component elements are all arranged around a central point. I call this class of designs *circular grids.*

Analyzing mason marks

Let's go through the exercise of analyzing the mason mark designs to illustrate the usefulness of Logo shape and placement notation. The following discussion is not the only description of what one can see in stone mason marks; don't be alarmed if the work you did yourself doesn't correspond exactly to the following illustrations. There is no single "correct" answer. But there is a proper style of approach. The following examples are designed to show you that style.

Go back and look carefully at all the stone mason marks at the end of the last chapter. Think about the following questions:

1. What shapes are used?
2. What shapes are not used?
3. What shapes are used together?
4. What shapes are not used together?
5. Are individual marks symmetric? If yes, can you see the point of symmetry? Are some marks partially symmetric? (Be sure that you have a definition of symmetry at your fingertips. Are there different kinds of symmetries?)
6. Within any individual mark, is any one shape repeated over and over again in a way that might suggest recursion?
7. What are the characteristics that make all the marks alike?
8. What are the characteristics that make each mark unique?
9. What already written Logo procedures could be used to produce parts of the marks?
10. What new Logo procedures need to be designed?

Now look back over the marks and quickly jot down your answers to these questions. Don't mull over the words; write them down as they come to you.

Written observations

You probably noticed that

1. All the designs are composed from a few basic shapes: circles, squares, and triangles. You can produce all of these easily with CNGON.

2. All the marks reveal a common characteristic: the orientation of the polygons around a center. Each mark has a design symmetry "around" the center of this theme circle. Each mark uses a limited combination of polygons. For example, squares and circles occur together in the same mark, but triangles and squares do not. Excepting the circle, no polygon beyond the 4-gon level is used. There are, for example, no pentagons or octagons. For polygons centered on the circle's center, CNGON can be used.

3. Two polygons of the same size and shape are often overlapped to form stars. For example, one triangle in Mark 2 is rotated 60 degrees from another, to form a six-pointed star, while one square, in Mark 1, is rotated 45 degrees from another to form an eight-pointed star. This is easy: use a combination of CNGONs and RTs.

4. There are two special kinds of polygon placements within the marks that require special attention. First, the grid placement of squares in Marks 7 and 10; and second, the placement of polygons at the vertices of other polygons: for example, the circles at the vertices of the smallest triangle in Mark 13. You will need two new procedures here.

5. There are two forms of recursion that appear in the group of marks, and each of the marks has at least one of them operating: first, the grid placement of boxes already mentioned might be structured with a recursive procedure—the fact that Mark 7 has twice as many boxes as Mark 10 is "very suggestive" of

recursion; and second, notice the occurrence of polygons nesting inside circles nesting inside polygons. This nesting is called "inscribed" polygons. That is, a polygon is inscribed, or drawn, to fit within another shape. You will see this nested-shapes recursion in many of the marks: note the two smaller squares nested inside the larger squares in Mark 1; also note the same kind of nesting in the triangles of Mark 2. New procedures will be required here.

6. Straight lines are sometimes used to join the vertices of some of the basic polygons—the diagonal cross inside a square—or the polygons formed by the overlap of the basic shapes—the lines joining the intersection points of the two triangles. New procedures here, too.

You will need new procedures for the last three points:

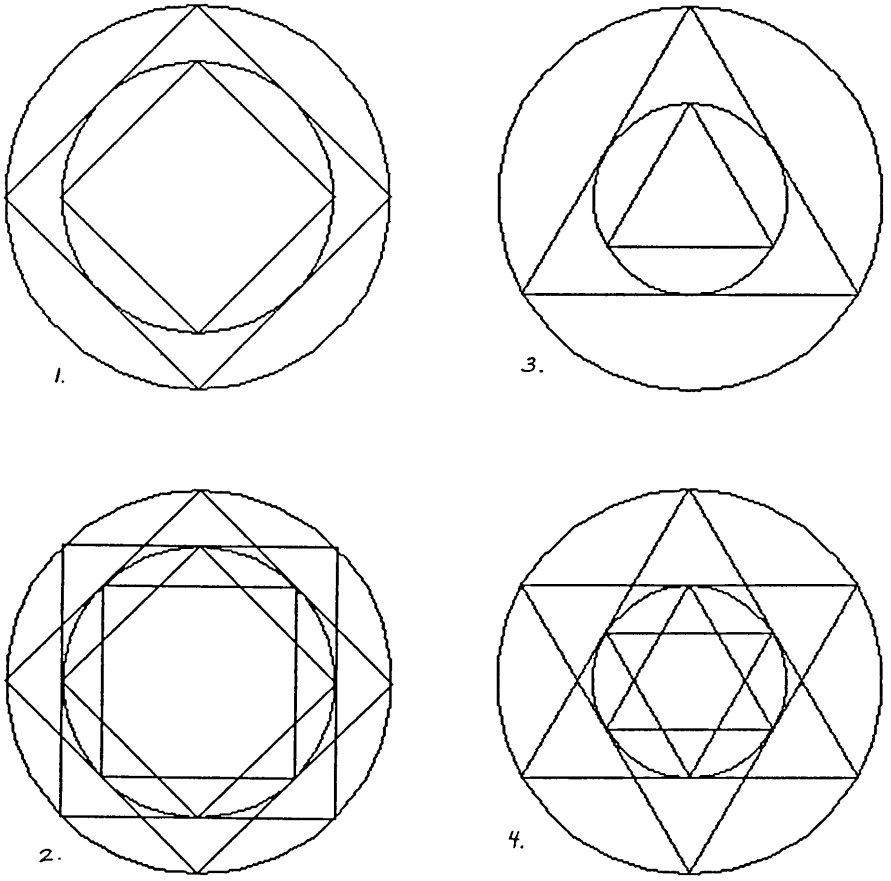
1. Placement of polygons at the vertices of other polygons.
2. Two kinds of recursion: nested/inscribed polygons and grids.
3. Some special line drawers to connect certain polygon vertices.

Recursive design features

Let's start with the recursion while your energy is high. Until recursion becomes second nature to you, you will need a lot of concentration to understand how it does what it does. But believe me, recursion will soon become as natural to you as using any other form of shape description.

Look at the following four designs. Then look back at Mark 1 and 2. See how natural some of these shape descriptors are becoming? Do you see the kind of recursive procedure that is needed, just by looking at these pictures?

Put your ideas into words, though, just to be on the safe side. Remember to record your thoughts and doodles in your notebook.

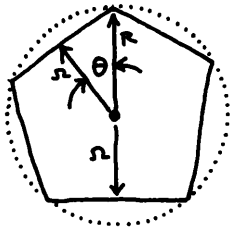


Inscribed polygons

We need a procedure that will first draw a circle of a given size, and then draw an NGON *inside* this circle. The NGON is said to be an inscribed polygon if it just fits inside a circle, and this will be the case when the NGON has the same radius as the circle. Next, the procedure needs to know if it should go to another level of inscription: should it draw a circle inside the NGON just drawn and inscribe another polygon inside this circle? Let's use :LEV to be the argument that indicates the number of nested polygons-inside-circles that is required. The procedure will have the following form:

```
TO INGON :N :RAD :C? :LEV
 ; INGON stands for inscribed NGON.
 ; :N is the number of sides of the inscribed polygon, and
 ; :RAD is its radius.
 ; Set :C? = 1 if you want the circles to be drawn.
 ; If :C? ≠ 1, no circles will be drawn.
 ; :LEV is the level of recursion.
 IF :LEV < 1 [STOP]
 ;  To stop recursion, as usual.
 IF :C? = 1 [CNGON 30 :RAD]
 ; Do you want the circumscribing circle drawn?
 CNGON :N :RAD
 ; Inscribe a polygon inside the circle.
 INGON  :N  (???)  :C?  (:LEV-1)
 ; Do INGON again with the correctly calculated :RAD and
 ; decremented :LEV.
END
```

The only problem is, what should go in the place of the `(???)`. This is the radius value of the next smaller inscribed polygon in terms of the polygon just drawn. The following little diagram should be useful as you think through this calculation.



$\theta = \frac{1}{2}(360/n) = 180/n$

$\cos\theta = \frac{r}{R}$

$\cos(180/n) = \frac{r}{R}$

Notice that a polygon has been drawn inside a circle of radius R. What is the radius of the circle inscribed inside this polygon? To answer this question, we must find an expression for r in terms of R. Then we can handle the `(???)` term in the `INGON` procedure.

Do you see that the angle θ (theta) equals one-half of 360/n, where n is the number of sides of the polygon? Using the definition of the cosine, we have cos θ = r/R. By putting these two expressions together and rearranging terms, we find the needed expression:

```
r = R*cos 180/n
```

We can now finish up INGON. The penultimate line becomes:

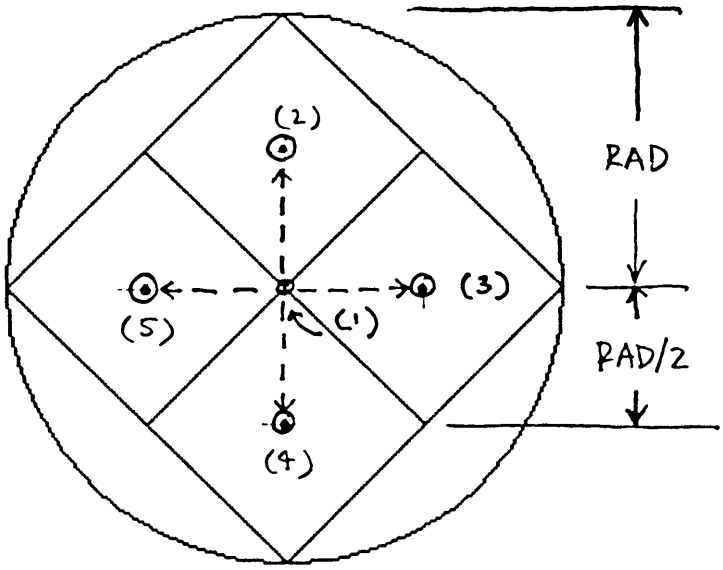
```
INGON :N (:RAD*COS 180/:N)  :C? (:LEV-1)
```

The underlined expression has replaced the `(???)`. OK? Let's try it out.

```
INGON 4 100 1 2
; Gives design 1 on page 120.

CG
INGON 4 100 1 2 RT 45
INGON 4 100 0 2
; Gives design 2 on page 120.

CG
INGON 3 100 1 2
; Gives design 3 on page 120.

CG
INGON 3 100 1 2 RT 60
INGON 3 100 0 2
; Gives design 4 on page 120.
```

Grid-of-squares turtle walk

Now for the grid of squares that appears in Marks 3, 6, 7, 10, and 16. Let's go through a turtle-walk exercise similar to that introduced to solve the CNGON problem at the end of Chapter 1. Imagine that you are at the center of a circle of radius :RAD and that you want to fill up an inscribed square with other squares. How would you go about doing it?



1. Starting at point 1, draw a `CNGON 4 :RAD`. The radius of this square equals the radius of the circle in which you are sitting.

2. Next, consider dividing this square into four smaller squares. You could draw these four squares by first going forward, after picking up your pen, by `:RAD/2`, putting down your pen at position (2) and running a `CNGON 4 :RAD/2`. Then, get yourself back to point (1) with your pen up. Turn right 90 degrees and get over to position (3). Run another `CNGON 4 :RAD/2` and get back to the center point (1). Do the same for (4) and (5)

3. Ready for the recursion twist? Suppose you want to divide the square centered on position (2) into four smaller squares. On arriving at point (2), you would want to draw four squares around this spot. You must go forward by an amount equal to half the distance `:RAD/2`, do a `CNGON 4 (:RAD/2)/2`, and then get back to (2). You would then turn 90 degrees in preparation for the second square of radius `(:RAD/2)/2`. The remaining two squares would follow in similar fashion.

4. But wait. Suppose when you were in position to draw the CNGON 4 with the radius (:RAD/2)/2, you considered yourself to be in the center of a square that should be divided again into 4 smaller squares. This is recursion.

And here is the Logo version of the words above:

```
TO SQUARES  :SIZE  :LEV
 ; Grid maker of squares in groups of four.
 IF :LEV < 1 [CNGON  4  :SIZE  STOP]
 ; Draw a square if you wish to recurse no deeper.
 REPEAT 4  [PU  FD  :SIZE/2  PD -
            ;  Get to center of a square.
            SQUARES  (:SIZE/2)  (:LEV-1) -
            ; Consider current position as center of 4
            ; squares.
            PU  BK  :SIZE/2  PD  RT 90]
            ; Get back to center and turn 90 degrees in
            ; preparation for moving to the next center of
            ; squares.
END
```

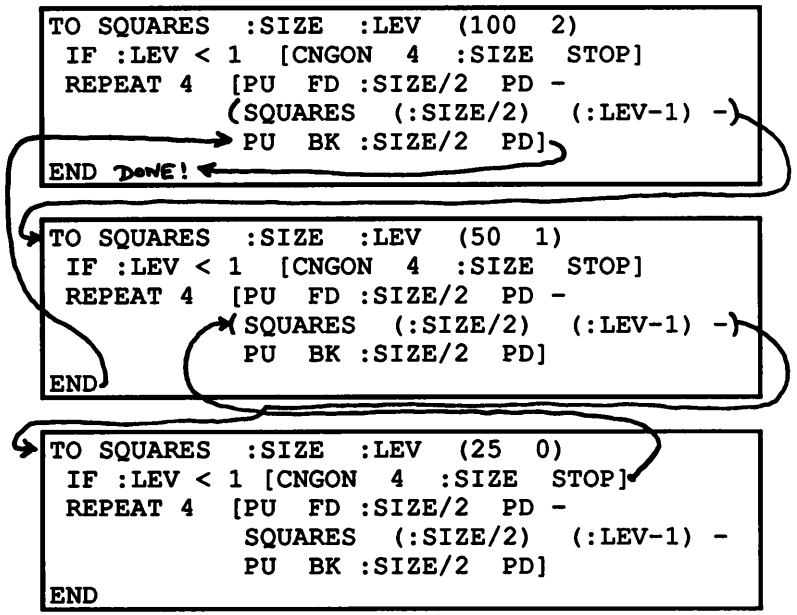
Another recursion diagram

```
STEP
SQUARES 100 2
```

Watch the turtle read each instruction and then carry it out. The STEP command slows the action of the turtle so that you can watch both the screen and the procedure. Use the following diagram to follow how the different levels of recursion are linked. This recursion diagram is similar to the one you saw when you worked on RECGON. Remember? If you have trouble comparing what you see on the screen with this recursion diagram, turn your printer on to print out what Logo is doing. Compare this printout, line by line, with the recursion diagram.

SQUARES recursion diagram

The following recursion diagram describes what "happens" after you type SQUARES 100 2. The level of recursion here is 2. Don't forget to type STEP.

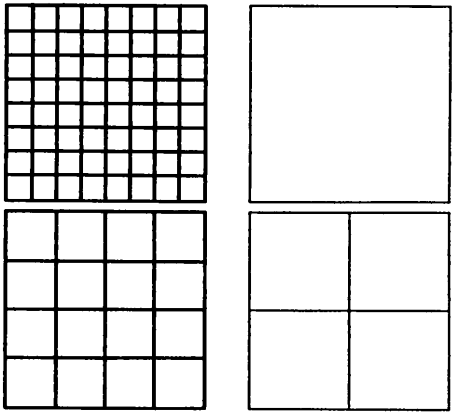


Exploring recursing squares

Now, design an EXPLORE-type procedure to investigate how SQUARES operates at different levels of recursion. The following is such a procedure.

```
TO SQ.EXPLORE :SIZE :L1 :L2 :L3 :L4
 ; To explore the SQUARES procedure at
 ; different recursion level and size values.
 PU SETXY 70 60 PD
 SQUARES :SIZE :L1
 PU SETXY 70 -60 PD
 SQUARES :SIZE :L2
 PU SETXY -70 -60 PD
 SQUARES :SIZE :L3
 PU SETXY -70 60 PD
 SQUARES :SIZE :L4
END
```

`RT 45 SQ.EXPLORE 80 0 1 2 3` produces the following diagram:



More complicated grids

Suppose you would like the `SQUARES` procedure to do something more complicated to the squares it gets around to drawing at the lowest recursion level. An example might be to put a cross in each box. The following procedure draws such a cross, but how should it be incorporated into `SQUARES`?

```
TO CROSS :SIZE
 REPEAT 4 [FD :SIZE BK :SIZE RT 90
END
```

We will want the cross made just after a square is drawn, so that they both use the same size value. That means that `CROSS` is placed just after `CNGON 4 :SIZE` in `SQUARES`. The following revised `SQUARES` procedure, called `SQUARES.X`, incorporates `CROSS`. Changes from or additions to the `SQUARES` procedure are underlined. You might guess that any number of alternative procedures could replace `CROSS`. A circle drawer `(CNGON 30 :SIZE)`, for example.

```
TO SQUARES.X :SIZE :LEV
 IF :LEV < 1 [CNGON 4 :SIZE CROSS :SIZE STOP]
 REPEAT 4  [PU FD :SIZE/2 PD -
            SQUARES.X (:SIZE/2) (:LEV-1) -
            PU BK :SIZE/2 PD RT 90]
END
```

The diagrams on the next page are examples of fancy grid recursion. They exhibit some of the flavor of stone mason marks as well. You should be able to reconstruct them yourself.

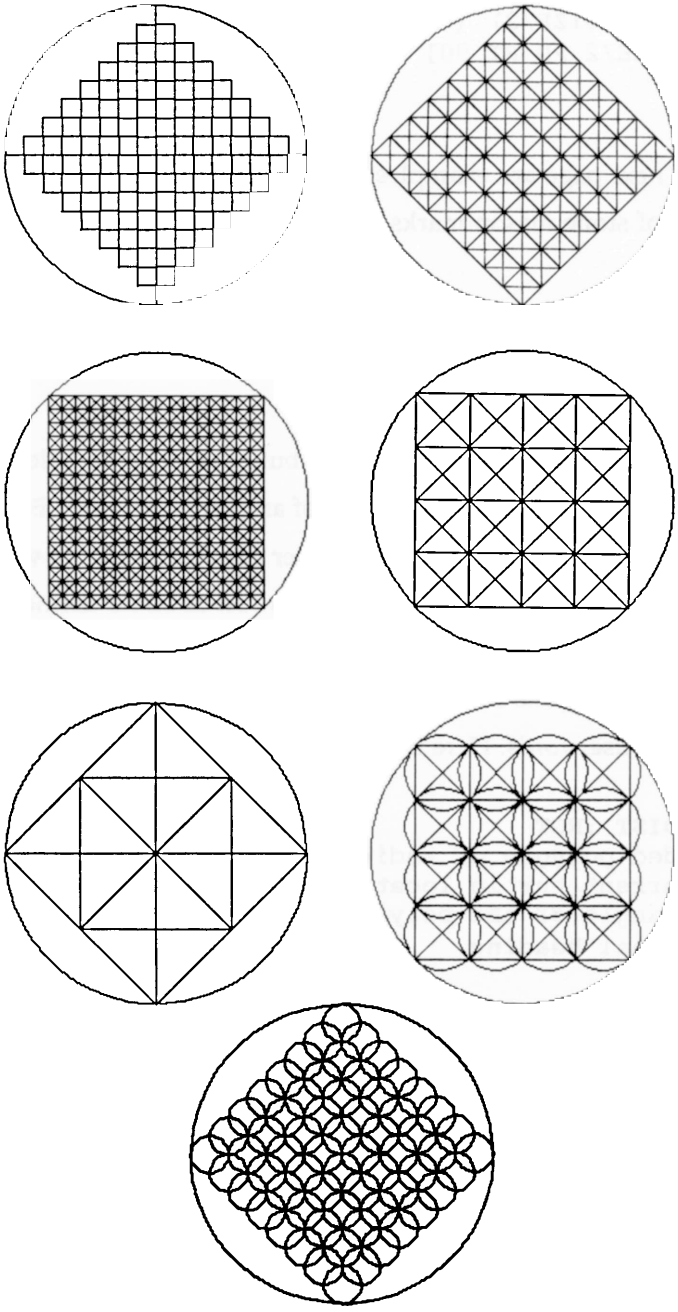
Placement of polygons on other polygons

RECGON was one procedure for doing this kind of thing, but now let's assemble a procedure that will put any kind of polygon at the tips of any other polygon. But maybe we can break this job down into something simpler than RECGON. All we really need is a procedure that will place and orient a CNGON at some defined distance from the turtle's current location and then return the turtle to its starting place. It's getting easier to use the notation of Logo to talk about a new shape placement than it is to use words. Here it is:

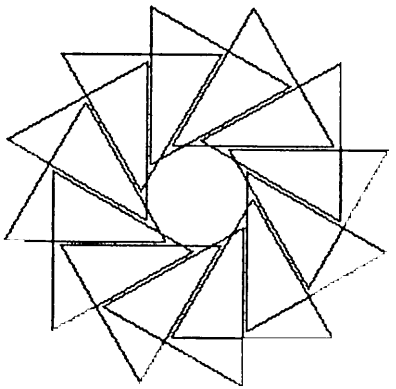
```
TO TIPGON :N :RAD :DIST :ROT
 ; To place an n-sided polygon of radius :RAD, at a distance
 ; :DIST from the current turtle location.
 ; :ROT is rotation angle of the polygon is relation to
 ; the turtle's original heading.
 PU FD :DIST RT :ROT PD
 CNGON :N :RAD
 PU LT :ROT BK :DIST PD
END
```

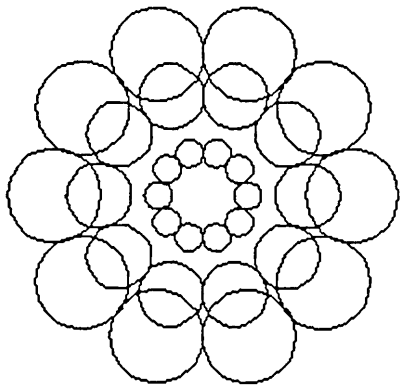
TIPGON is a building block procedure that fits nicely into others. On the page after next are a few examples of how TIPGON can be so used. You should be able to reconstruct all of these quite easily. By inspection?

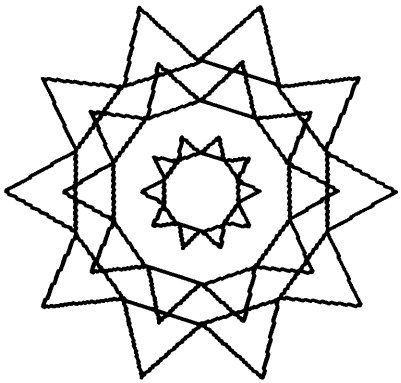
Mason mark designs from squares

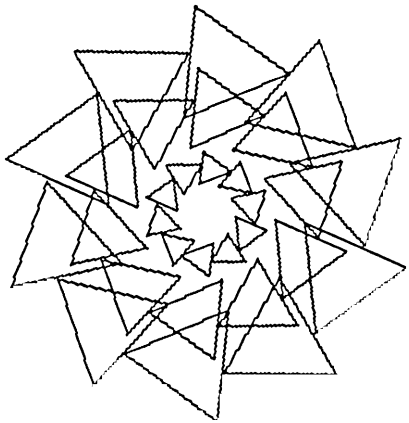


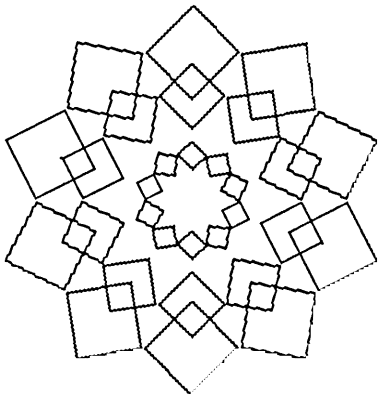
Some TIPGON-generated figures











Special line procedures

Let's overlap two similar NGONs to form a regular star with 2*:N points. The trigonometry needed to find the rotation between the two NGONs is easy: the angle between points of the composite star will be 360 divided by the number of points or 360/(2*:N). So the star operation is:

```
CNGON :N :RAD
RT 360/(2*:N)
CNGON :N :RAD
```

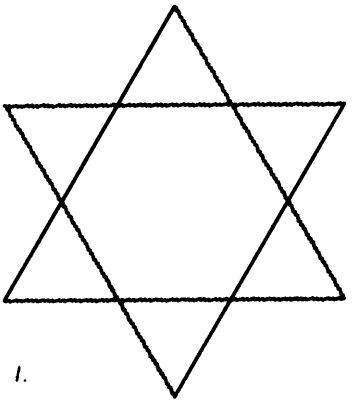
An example. Make a six-pointed star from two triangles of radius 100:

```
CNGON 3 100
RT 360/(2*3)
CNGON 3 100
```

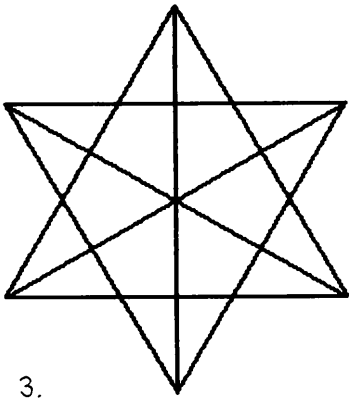
The star produced is top left one on the next page.

The bottom left figure is the star with the intersection points of the two triangles connected. The top right figure shows the star again, but now with the star points opposite each other joined. The bottom right star combines the connections of the top right and bottom left ones.

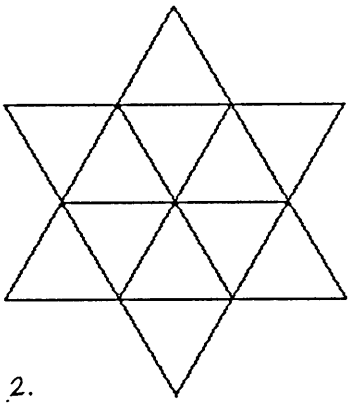
The exercise for you is to think about the trigonometry needed for this exercise and then to design a procedure that will draw these two kinds of lines for any star drawn from a composite of two similar polygons. Try out your own ideas before going any farther. Here is my solution; I've called the procedure STRIPES:



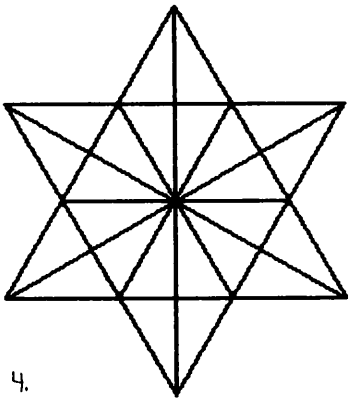
1.



3.



2.



4.

```
TO STRIPES :N :RAD :L1? :L2?
 ; :N is number of ngon's sides and :RAD is radius.
 ; If :L1? = 1 draw lines joining the NGON intersection
 ; points.
 ; If :L2? = 1 draw lines joining opposite star points.
 IF :L1? = 1 [RT 90/:N
              REPEAT :N [CNGON 2 -
              (:RAD*COS 180/:N)/(COS 90/:N) -
                        RT 180/:N] -
              SETH 0]
 IF :L2? = 1 [REPEAT :N [CNGON 2 :RAD -
                         RT 180/:N] -
              SETH 0]
END
```

You might want to verify that STRIPES works in a number of situations. Here are two examples from me. The bottom left star was generated by:

```
CNGON 3 100
RT 360/(2*3)
CNGON 3 100
STRIPES 3 100 1 0
```

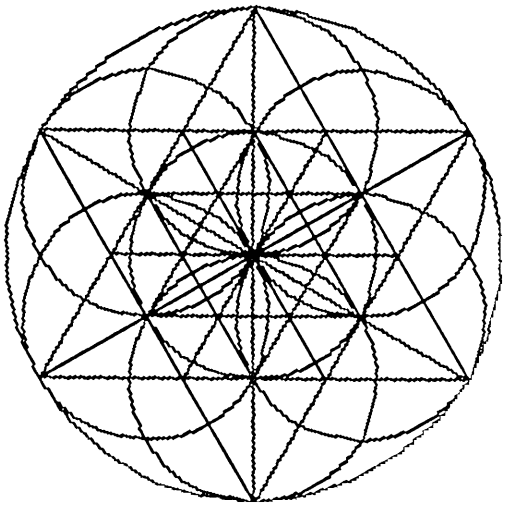
The top right star came from this:

```
CNGON 3 100
RT 360/(2*3)
CNGON 3 100
STRIPES 3 100 0 1
```

The bottom right star combines the top right and bottom left stars.

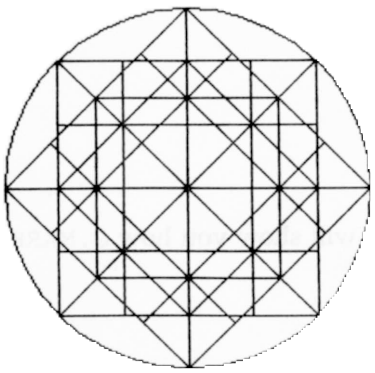
That's all. You should now be able to generate most of the stone mason marks given at the end of Chapter 3 using the procedures that have been discussed in this chapter. Want to try a few?

```
TO MARK.13 :SIZE
 INGON 3 :SIZE 1 2
 RT 180 (INGON 3 :SIZE 0 2) LT 180
 REPEAT 6 [TIPGON 30 (:SIZE/2) (:SIZE/2) 0
           RT 60]
 STRIPES 3 :SIZE 1 1
END
```



Here's another. Notice that this mark uses both the grid recursion and the inscribed polygon recursion patterns:

```
TO MARK.16 :SIZE
 CNGON 30 :SIZE
 REPEAT 2 [SQUARES :SIZE 2 RT 45]
 STRIPES 4 :SIZE 0 1
 REPEAT 2 [INGON 4 :SIZE 0 2 RT 45]
END
```



Generalized Gothic stone mason mark procedures

The procedures we have just looked at, `MARK.13` and `MARK.16`, are not very general. They can each produce only *one* design. Their only generality is an ability to produce fixed designs of different sizes. That isn't very interesting.

What about designing a more general mark procedure that would include an additional argument for *controlling the shape* of the design produced? Exercise 4.6 will ask you to design a generalized mason mark procedure with at least 3 arguments. Here is a preview of such a generalizing approach.

The following procedure was designed to draw either Mark 1 or Mark 2. I also wanted to see a mason mark figure based on pentagons and hexagons. I therefore added a second argument, `:N`, that would control the shapes selected to construct the mark. This new procedure thus has an argument that determines the overall shape as well as one for setting the size of the mason mark.

It is often interesting to see what "happens" to designs produced by a Logo procedure as the procedure's arguments are given more and more extreme values. Sometimes designs produced with such extreme value arguments are very boring. But other times one finds designs that are totally unexpected and very exciting.

Here is a start at my generalized mason mark:

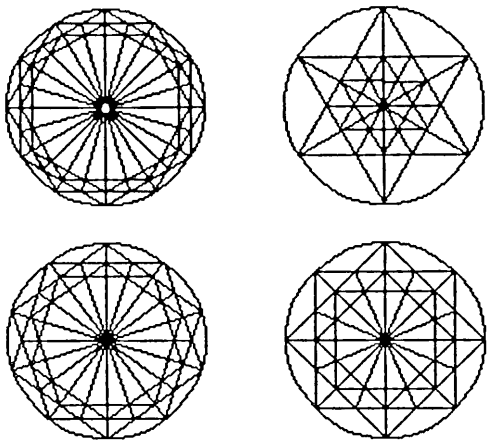
```
TO G.MARK :N :SIZE
 CNGON 30 :SIZE
 INGON :N :SIZE 0 2
 RT 360/(2*:N)
 INGON :N :SIZE 0 2
 LT 360/(2*:N)
 STRIPES :N :SIZE 1 1
END
```

And here is an EXPLORE-type procedure that will show you how G.MARK-produced design change with changing :N values:

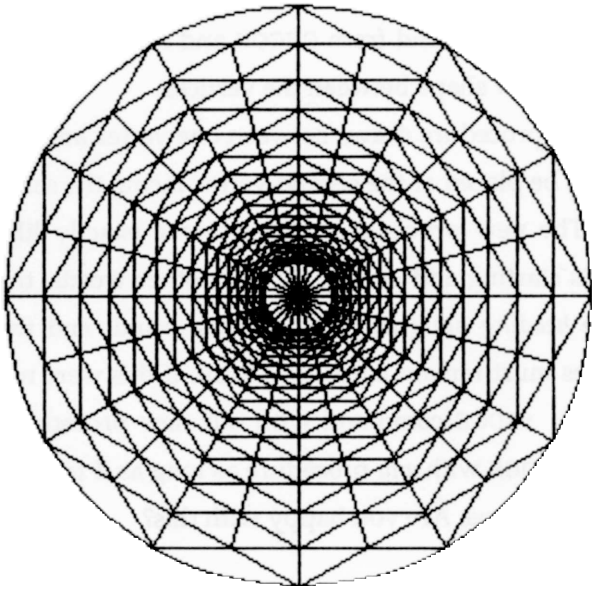
```
TO M.EXPLORE :SIZE :N1 :N2 :N3 :N4
 ; To explore the G.MARK procedure at different :N values.
 PU SETXY 70 60 PD
 G.MARK :N1 :SIZE
 PU SETXY 70 -60 PD
 G.MARK :N2 :SIZE
 PU SETXY -70 -60 PD
 G.MARK :N3 :SIZE
 PU SETXY -70 60 PD
 G.MARK :N4 :SIZE
END
```

The following diagram is produced with

```
M.EXPLORE 50 3 4 5 6.
```

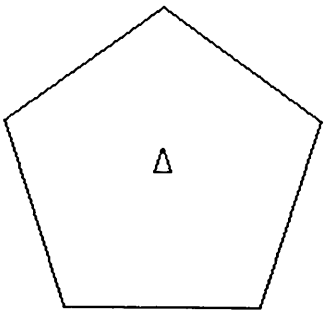


Comment: the extreme values, `:N = 6`, for example, aren't very interesting. The figure becomes confused and muddied with too many lines. Here is the reason why all of the stone mason designs in the illustrations given at the end of Chapter 3 limited their ngons to a max of n = 4, squares. Circles seem to be a special case. The `G.MARK` based on the hexagon above does suggest some work in the following direction. Spider webs, maybe?



The quality of polygon lines

Here is a simple pentagon produced by the CNGON procedure. Look at the five edges of this polygon. They are straight lines and they connect the vertices of the pentagon together.



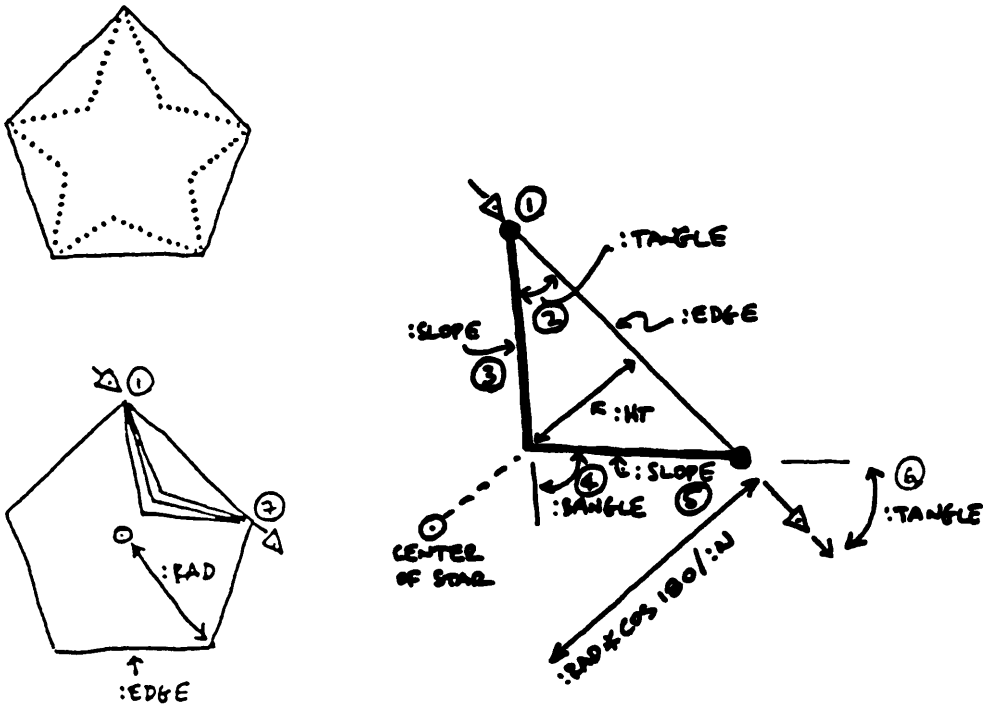
Must we use a straight line? Suppose we want the vertices to be connected with a line whose quality is different from that of a straight line. Different line qualities could mean wavy, jagged, or dotted lines. Or maybe we might decide that we want the vertices of a polygon to be joined together with a special shape. Why not remove the FD edge command from CNGON and replace it with some other procedure that draws a new shape between the vertices.

Once whatever shape the new procedure draws between one vertex and the next is completed, the turtle must be placed at the second vertex facing the same direction it faced at the first. The new procedure must operate exactly like FORWARD in terms of the turtle's starting and ending state. In other words, the turtle must be left in the same *heading state* in which it was found, and the distance between the two vertices must remain equal to :EDGE. If this were not true, we could not replace the FD :EDGE command with our new procedure. Why? In fact, we can replace FD :EDGE with *any* procedure as long as it follows FORWARD's starting and ending conditions. Are you happy with this?

Can you state the conditions necessary for us to replace one part of a procedure with an alternative part? How must the parts relate? How is this business like the notion of state transparency?

Stars

Let's consider drawing stars. Look at the sketch below. A star is a polygon with a different quality of edge. Instead of a straight-line edge, the star has an edge with a "kink" in it. If we could write a procedure to draw "kinked" edges that follow the beginning and ending state conditions of FD, we could exchange the FD :EDGE in CNGON with the kinked line procedure. This exchange will create a new gon-drawing machine called STARGON. If we are careful, STARGON will be a more general form of CNGON because it will draw stars as well as polygons.



Let's call the new procedure that will draw the kinked line STAREDGE. Before we do a turtle-walk description of the staredge, let's decide how we will characterize its major feature, kinkiness, in terms of an argument called :KINK. Suppose we say that if :KINK = 0, there will be no kinkiness to the staredge and so the star drawn will look exactly like a polygon. That's fine: a star with no kinkiness *is* a polygon. On the other hand, if :KINK = 1, then the kink in the staredge will extend down to the center of the figure. The shape of this star will look like the spines of an umbrella. A :KINK of .5 should produce something midway between a polygon and an umbrella frame.

Turtle walk through the staredge

1. The turtle begins at position (1). It is facing the next vertex at position (7).

2. The first thing that STAREDGE must do is to turn right by the angle labeled :TANGLE (for top angle) at (2). Don't worry about the calculations yet; we can work those out later.

3. Now, go forward by the amount labeled :SLOPE. (3)

4. Now, turn left by the angle labeled :BANGLE (for bottom angle) at (4).

5. Go forward by the :SLOPE. (5)

6. Turn right :TANGLE. (6)

7. The turtle is now in position (7). STAREDGE has moved the turtle from vertex 1 to 7, and the turtle heading at 7 is the same as it was at 1. This corresponds to what FORWARD accomplished, so we can exchange the two procedures.

Staredge calculations

Here are the calculations that will be needed. I've written a supporting procedure called SQ that squares its argument; your own version of Logo may already have this command. SQRT is my Logo's command for square root.

I am pleased that this project will encourage you to review a little more geometry and trig. You will need to remember the arctangent and its associated ARCTAN command; we use this to calculate the angle :TANGLE. Also, you might recall the Pythagorean theorem: in a right triangle, the square of the length of the hypotenuse equals the sum of the squared lengths of the other two sides. We use that to find the :SLOPE.

```
TO SQ :A
 ; To square a number.
 OP :A*:A
END
```

Calculations

```
:EDGE    =  2*:RAD*SIN 180/:N
:HT      =  :KINK*:RAD*COS 180/:N
:TANGLE  =  ARCTAN :HT :EDGE/2
:SLOPE   =  SQRT (SQ (:EDGE/2) + SQ :HT)
:BANGLE  =  2*:TANGLE
```

Putting STARGON together

Where shall we carry out these calculations? We could place them inside the new procedure STAREDGE. But this wouldn't be very efficient since the math needs to be done only *once,* not every time a STAREDGE is drawn. So let's place the calculations at the start of the procedure STARGON. Here are the two new procedures.

```
TO STAREDGE :TANGLE :SLOPE
 RT :TANGLE
 FD :SLOPE
 LT 2*:TANGLE
 FD :SLOPE
 RT :TANGLE
END

TO STARGON :N :RAD :NOTCH
 ; Centered star exercise.
 (LOCAL "EDGE "HT "TANGLE "SLOPE)
 MAKE "EDGE 2*:RAD*SIN 180/:N
 MAKE "HT :NOTCH*:RAD*COS 180/:N
 MAKE "TANGLE ARCTAN :HT :EDGE/2
 MAKE "SLOPE SQRT (SQ (:EDGE/2) + SQ :HT)
 ; These MAKE commands calculate the values that STAREDGE
 ; will use below.
 PU FD :RAD PD
 RT 180 - (90*(:N-2)/:N)
 REPEAT :N [(STAREDGE :TANGLE :SLOPE) RT 360/:N]
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
END
```

Global and local variables

Notice that I have made the staredge variables all *local* variables. We don't need the value of these staredge calculations outside of STARGON. Do you remember the difference between local and global variables? *Global* variables have an existence *outside as well as inside* the body of the procedure that created them. These variables never go away, in fact, until they are erased. The command NAMES lists all global variables currently in your workspace and EDNAME allows you to edit a specific variable. Check your Logo language manual to verify these commands in your own Logo dialect.

Local variables, on the other hand, exist *only inside* the procedure that defines them. A local variable is, however, available to any procedure that is used within the body of the procedure that defined the variable.[1]

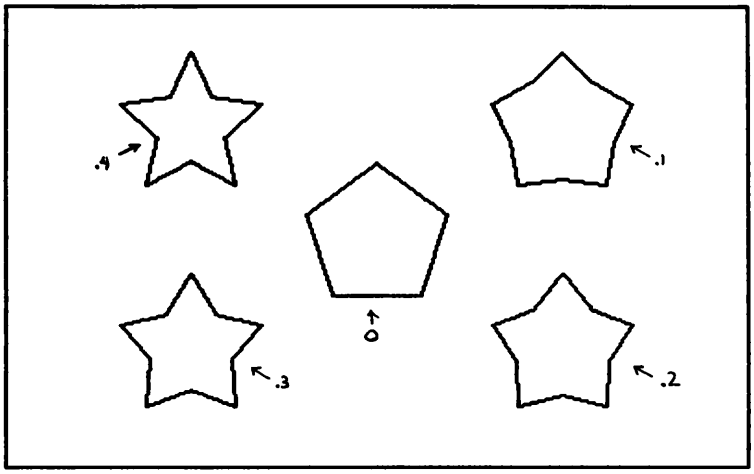
1. Most traditional Logos, including the dialect used in this book, apply this approach, but some recent implementations such as Coral Object Logo favor an

Logo isn't very good at "number crunching." Because Logo is so slow with mathematical calculations, structure your procedures to calculate as few times as possible. Group the "dirty" math work in one place using MAKE commands, so Logo can go on to what it does most quickly: moving the turtle about the screen. If possible, LOCALize all the MAKE-created variables.

A look at STARGONs with different :KINK values

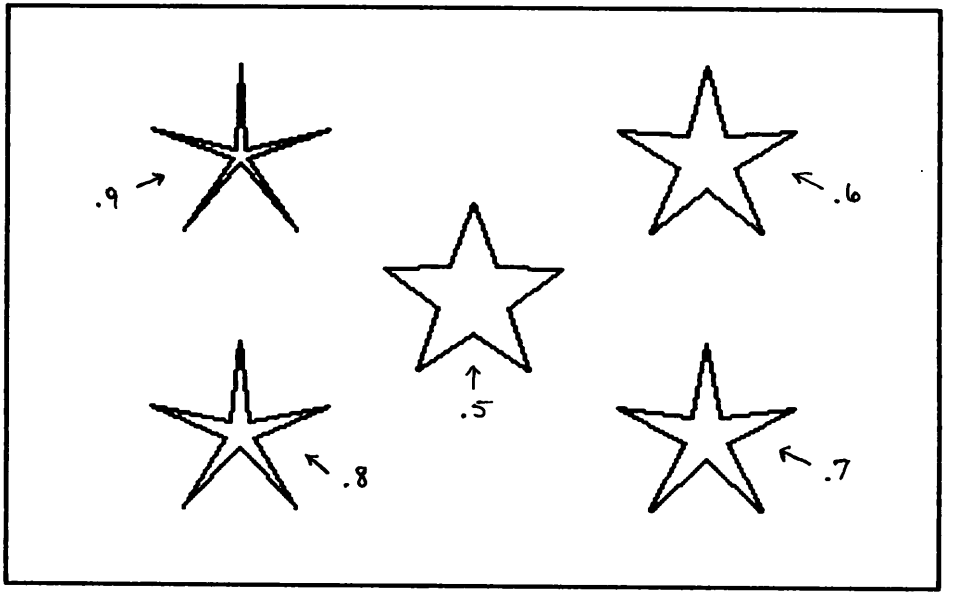
My recommended way to explore a new procedure is to design an EXPLORE-like procedure. Each of the following diagrams shows a set of five stargons. The value of the :KINK factor is noted on each individual stargon.

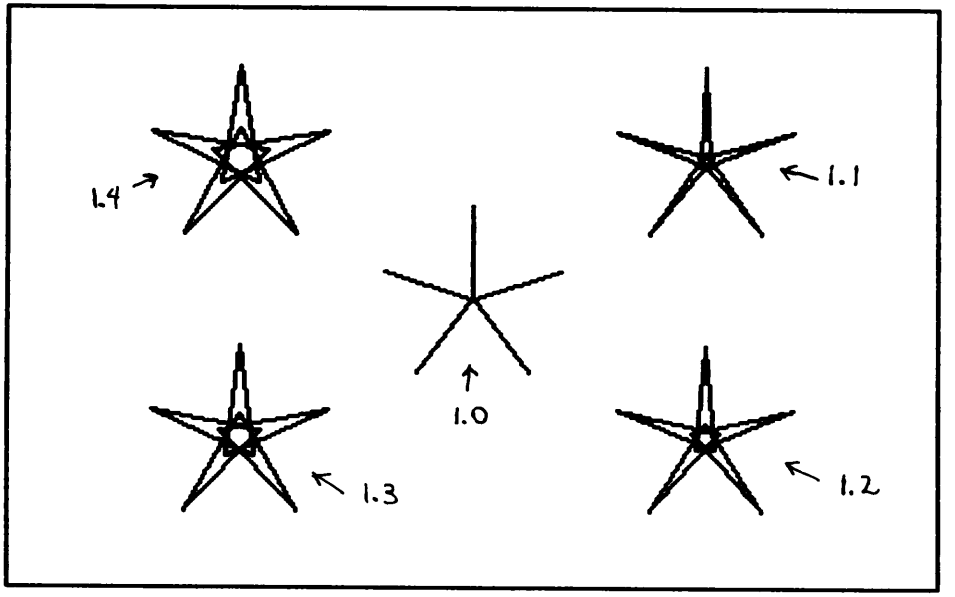
Note that the stars produced with a kink value greater than 1 are pulled "inside-out." Why is this? Look carefully. Perhaps you would like to use the STEP command to slow the turtle down as it draws one of these inside-out things.



---

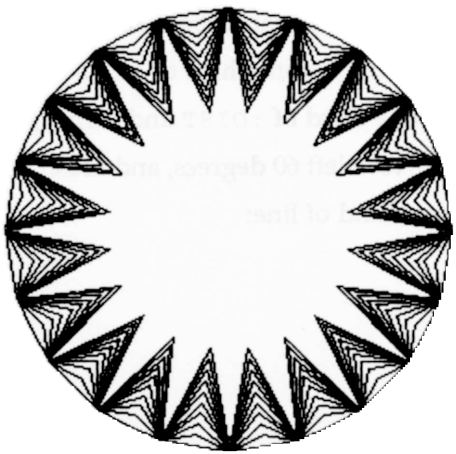
alternative approach in which local variables are available only to the procedure that created them and not to any others. Your Logo manual should describe the method it uses to define the range or "scope" of local variables. If it applies the second approach, you will need to make some (relatively simple) modifications to the procedures given in this book.

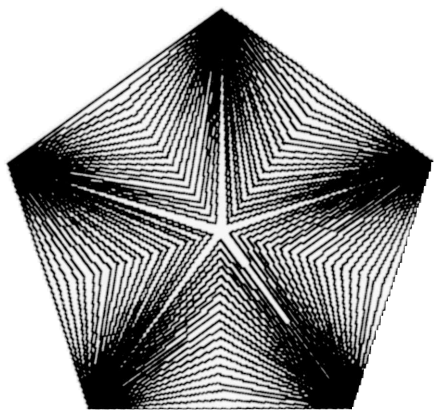

.9
.6
.5
.8
.7

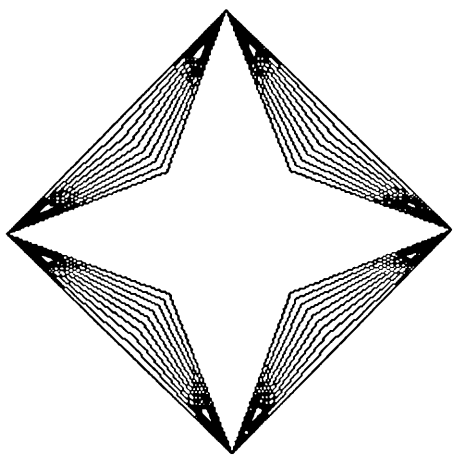

1.4
1.1
1.0
1.3
1.2

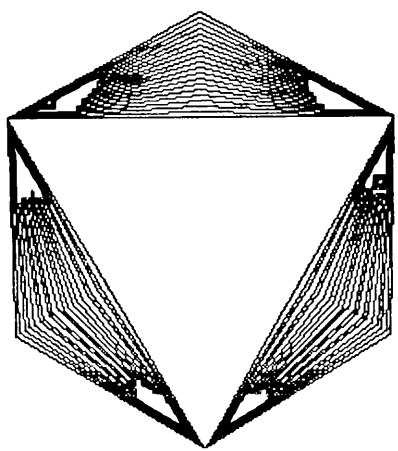
STARGON-produced designs

Below are several designs that explore the :KINK argument effect on STARGON-produced shapes. I won't give you the actual procedure used to create these designs, but it has much in common with CONGON of Chapter 3. Remember?







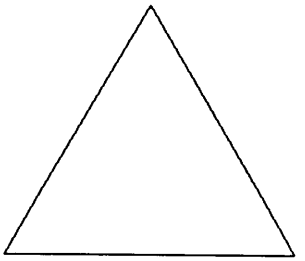


Fractalgons

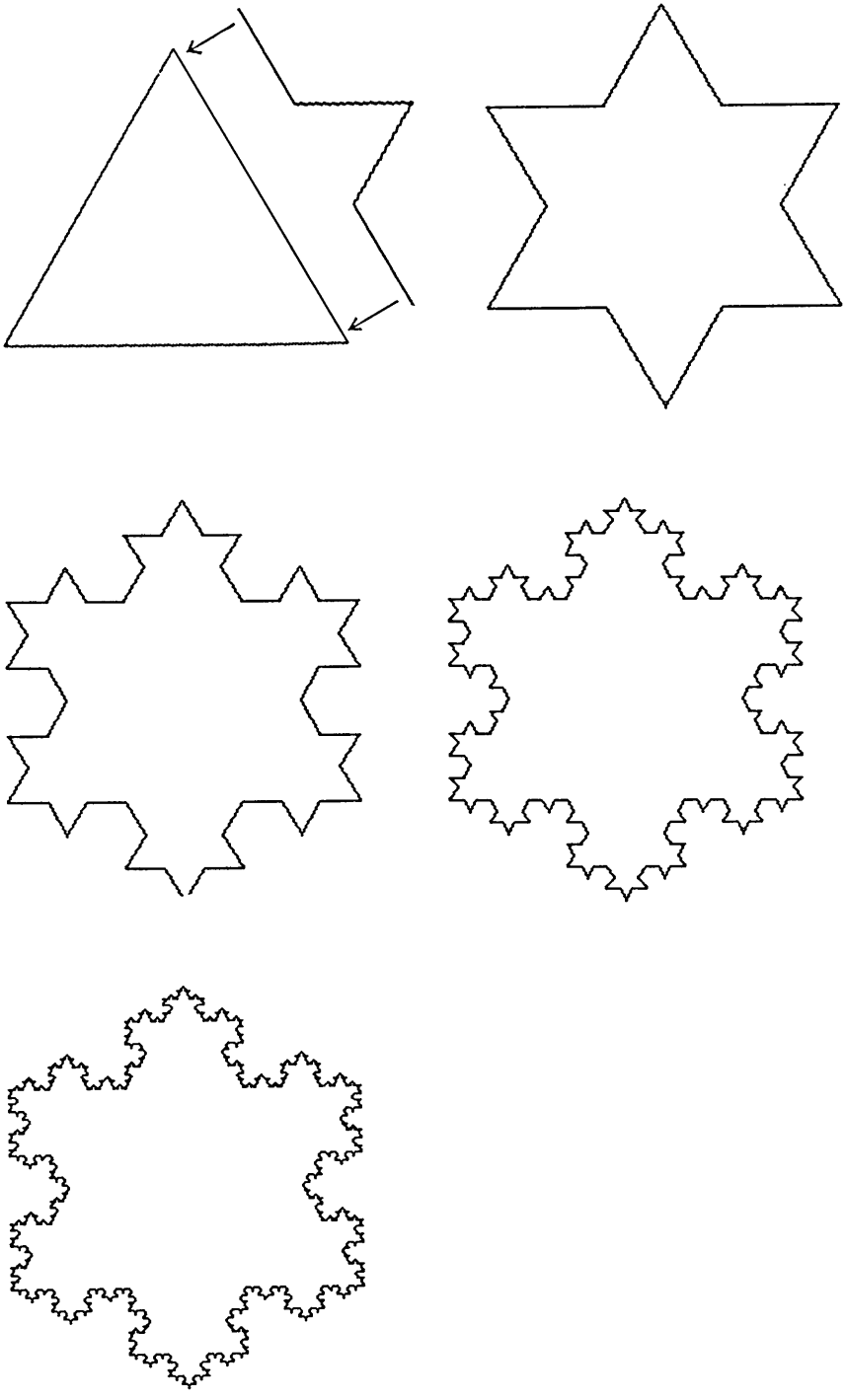
Put off for a moment asking what a fractal might be. Instead, consider yet another way of altering the quality of the polygon edges drawn by CNGON.

Recall how we built STARGON from CNGON in order to draw kinked lines between polygon vertices. Here is a turtle-walk description of another line shape that can replace FORWARD in CNGON: take the distance between vertices and call it :DIST. Now have the turtle go forward by one-third of :DIST and turn left 60 degrees. Then go forward another one-third of :DIST and turn right 120 degrees. Now go forward another :DIST/3, turn left 60 degrees, and finish up with FD :DIST/3. Here is the shape of this new kind of line:

And here is a triangle with ordinary straight sides:



What if you now replaced the straight line sides with the new shape? And after you have done this, what if you could replace all the straight line elements within the new shape with a smaller version of the new shape? And then, what if you could replace the remaining straight line segments with smaller versions of the shape . . . What is this? Recursion, again.



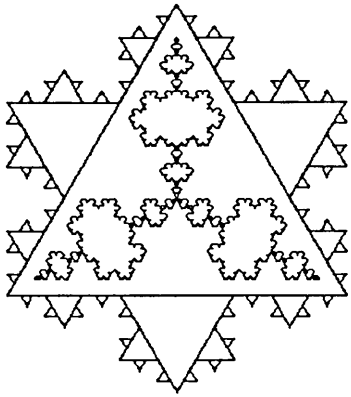
FRACTALGONs: a new recursion machine

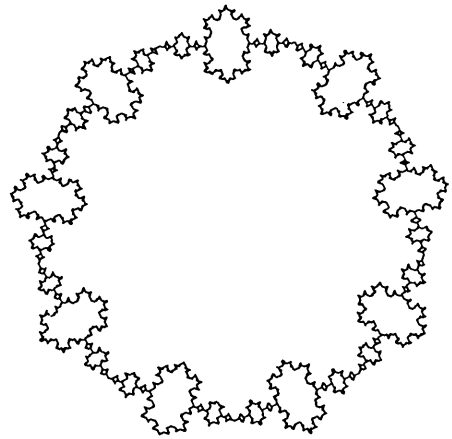
Look closely at the following two procedures. You should have a feeling for what is going on as soon as you scan the lines. But what will that argument :DIR do? Notice, too, that the kind of recursion used in FRACTAL is not exactly the same as that used in RECGON. There are four places within the procedure FRACTAL where FRACTAL "calls" itself (makes use of recursion). Make a recursion diagram for yourself and use STEP to verify that your recursion diagram arrows are correct.

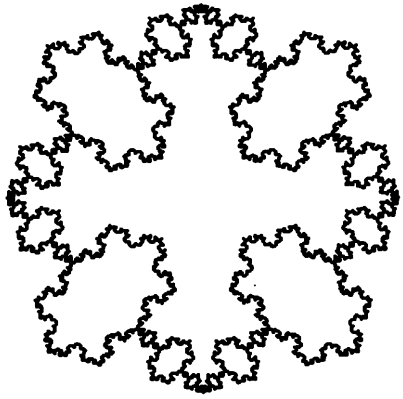
Can you now give a definition, based on FRACTALGON, of a fractal design? Of a fractal? Try. What is the quality of the edge line in a fractal design?

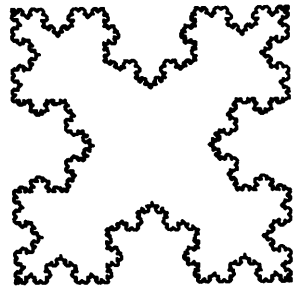
```
TO FRACTALGON  :N :RAD :LEV :DIR
 ; If :DIR = 1 turn outward.
 ; If :DIR = -1 turn inward.
 LOCAL "EDGE MAKE "EDGE 2*:RAD*SIN 180/:N
  PU FD :RAD PD
 RT 180-90*(:N-2)/:N
 REPEAT :N [FRACTAL :EDGE :LEV :DIR -
            RT 360/:N ]
 LT 180-90*(:N-2)/:N
 PU BK :RAD PD
END

TO FRACTAL :DIST :DEPTH :DIR
 IF :DEPTH < 1 [FD :DIST STOP]
 FRACTAL (:DIST/3) (:DEPTH-1) :DIR
 LT 60*:DIR
 FRACTAL (:DIST/3) (:DEPTH-1) :DIR
 RT 120*:DIR
 FRACTAL (:DIST/3) (:DEPTH-1) :DIR
 LT 60*:DIR
 FRACTAL (:DIST/3) (:DEPTH-1) :DIR
END
```

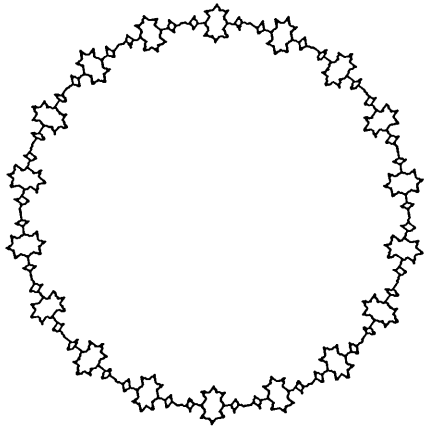
Some curious FRACTALGONs

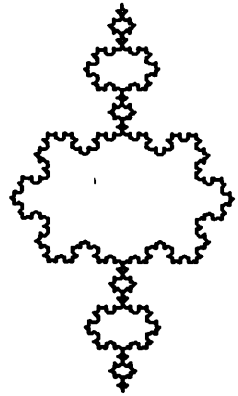












Observations

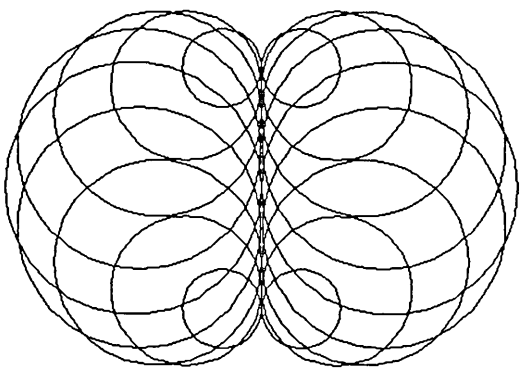
Have you noticed that most of the designs in the first three chapters have been "circular"? These circular patterns reveal a symmetry based on a central point, and that's why I've called them *circular grids.*

There is one final example of circular grids in the chapter. It is another example of how a complicated figure can be unpacked into its components and then restructured. This example shows how a generalized model of the basic component can lead to a wide variety of recomposed images based on the theme of the original. While all the images share the same original characteristics, they are surprisingly different in feeling.

Nephroids

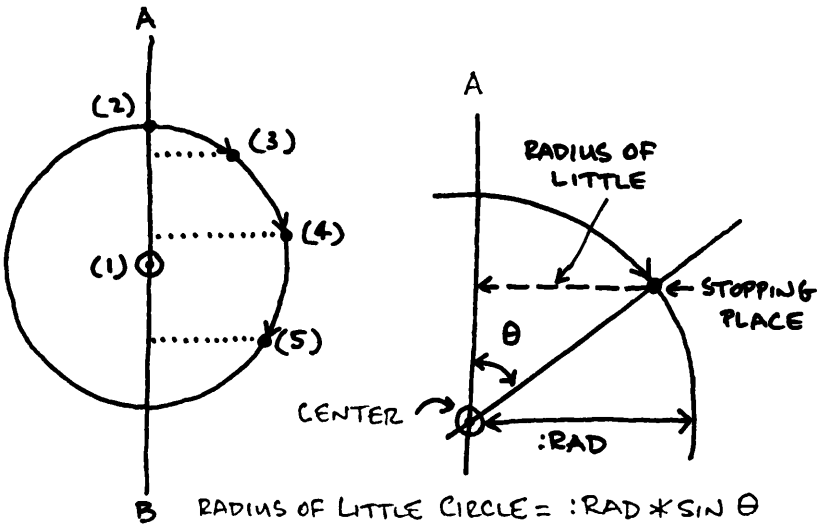
"Nephroid" means kidney-shaped. It is a nice example of a complex circular grid.

Look carefully at the following diagram. Can you decompose the design into its polygonal components as you did with the stone mason marks? Could you draw a nephroid on a drawing board with a straight edge and compass? Most important, could you do a turtle-walk scenario? That walk might begin to form in your mind as soon as you start scanning the design with your eyes . . .



Nephroid walk scenario

Here is a little sketch that might help. Imagine a circle whose center is labeled (1). Begin to walk along the circumference of this circle starting at point (2). Stop at equal intervals along the circumference (3, 4, 5, ...) and draw a circle—one at each stopping place—whose center is your current position. The radius of each circle will be such that its circumference will just touch line AB, the diameter of the big circle that you are walking on. How can you calculate the radius of each of the circles that you must draw?



Look at the right-hand diagram above. The thing that must be calculated is labeled "radius of the circle to be drawn." You know the `:RAD` of the big circle, and you could probably figure out the angle θ. If you know these two things, then trigonometry will again supply you with help. The radius you need is simply `:RAD*SIN` θ. The same simple relationships are useful over and over again.

We can use `TIPGON` to move out from the center (1) to a position on the circumference, draw the circle, and then move back to (1). You can then turn the turtle at (1) an appropriate amount in preparation for using `TIPGON` again.

What about that angle θ? You need that in order to calculate the radius of each of the circles that TIPGON places out there on the circumference of the big circle. You can use the command HEADING. HEADING gives you the turtle's current angular orientation as measured in degrees from the straight-up position. Straight-up is 0 degrees; pointing toward 3 o'clock is 90 degrees; straight-down is 180 degrees; and pointing toward 9 o'clock is 270 degrees. The line AB indicates the straight-up position. So HEADING will give you the angle you need to calculate the radius for each circle that must be drawn. What about tilting the design with respect to the line AB? Let :TILT represents the degrees of clockwise tilt.

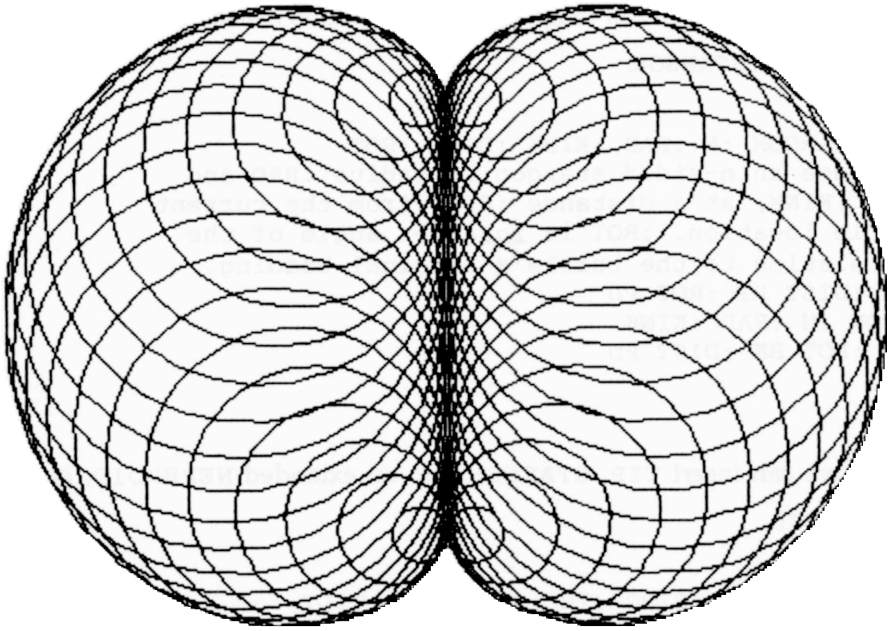
Here it is:

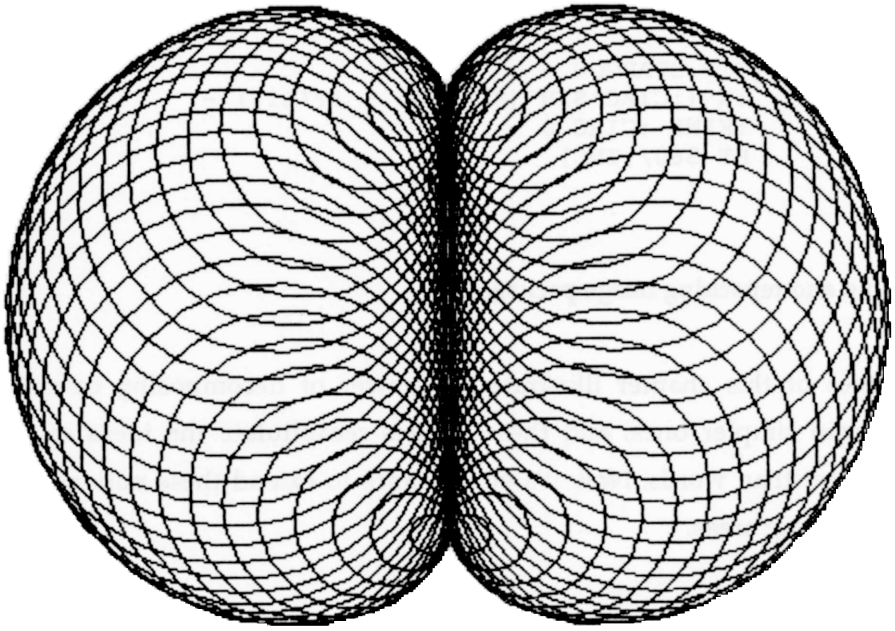
```
TO NEPHROID :CIRC :INC :TILT
 ; :CIRC is the radius of the big circle. :INC specifies
 ; how many times the turtle stops along the circumference.
 ; :TILT is clockwise rotation of figure from the
 ; horizontal.
 RT :TILT
 REPEAT :INC [TIPGON 30 -
              (:CIRC*SIN (HEADING - :TILT)) :CIRC 0 -
              RT 360/:INC]
 LT :TILT
END
```

On the next page are a few nephroids. Because the screen resolution is limited, these nephroids look as if they had been knitted rather than drawn.

Generalized nephroids

Let's consider producing a series of nephroid designs based on a more generalized model of the component part. Instead of circles, let's use STARGONs. With STARGONs we can still generate circle-based nephroids, but we can also explore nephroids built up from stars. And what will that look like?





First, let's extend TIPGON to manipulate STARGONs instead of CNGONs. Call the extension TIP.STARGON.

```
TO TIP.STARGON :N :RAD :KINK :DIST :ROT
 ; To place an n-sided stargon of radius :RAD and
 ; kink :KINK, at a distance :DIST from the current
 ; turtle location. :ROT is rotation angle of the
 ; in relation to the turtle's original heading.
 PU FD :DIST RT :ROT PD
 STARGON :N :RAD :KINK
 PU LT :ROT BK :DIST PD
END
```

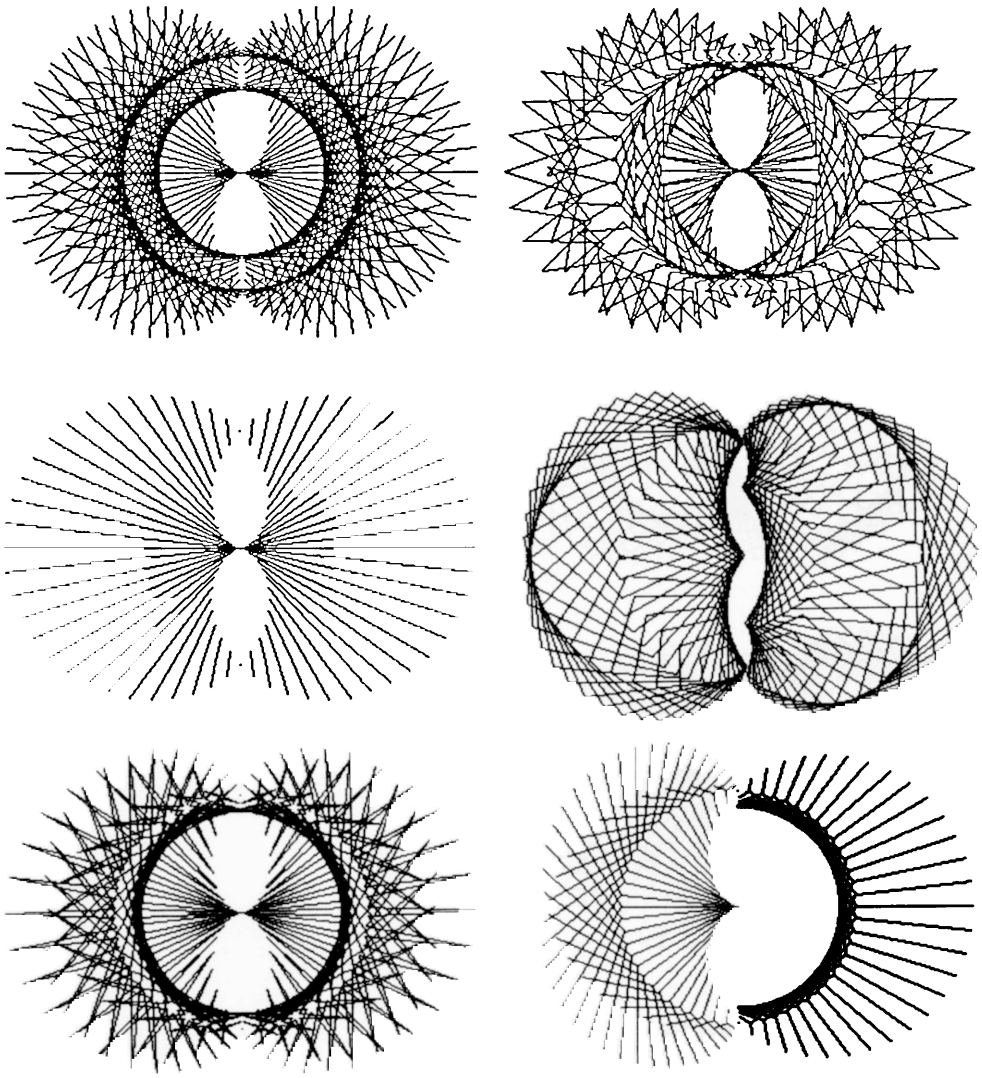
Now, we can insert TIP.STARGON into an extended NEPHROID, called STAR.NEPHROID.

```
TO STAR.NEPHROID :CIRC :INC :N :KINK :TILT
 ; :CIRC is the radius of the big circle. :INC specifies
 ; how many times the turtle stops along the circumference.
 ; :N is number of sides of component stargons with
 ; kink = :KINK. ;TILT is clockwise rotation of figure
 ; from the horizontal.
 RT :TILT
 REPEAT :INC [TIP.STARGON -
              :N (:CIRC*SIN (HEADING - :TILT)) -
              :KINK :CIRC 0 -
              RT 360/:INC]
 LT :TILT
END
```

Unpacking and repacking image packages

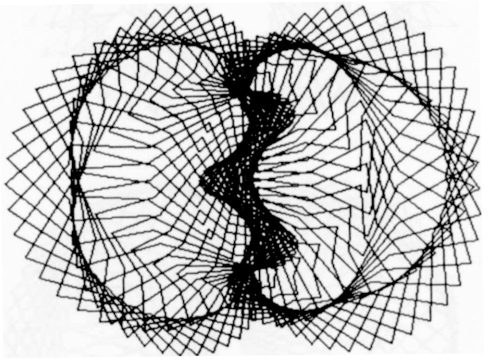
The images of this chapter illustrate the power of decomposing complex patterns into simpler ones, and these images recapitulate the ideas of the chapter better than words. Now it's your chance to extend these notions into more challenging areas.

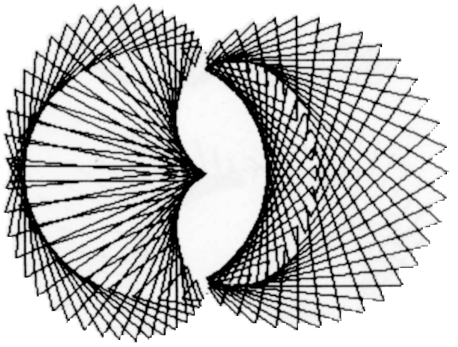
STAR.NEPHROIDs

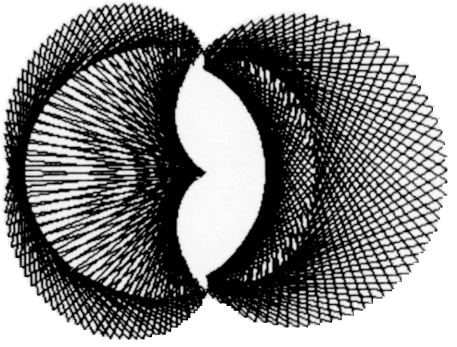


More STAR.NEPHROIDs

Some curious things seem to be happening. Some of these images are not symmetric. The left half is not the same as the right half. Why is it that some images close and others do not? If you have forgotten the term *closure*, look at the PIPEGONs of Chapter 2.







Exercises

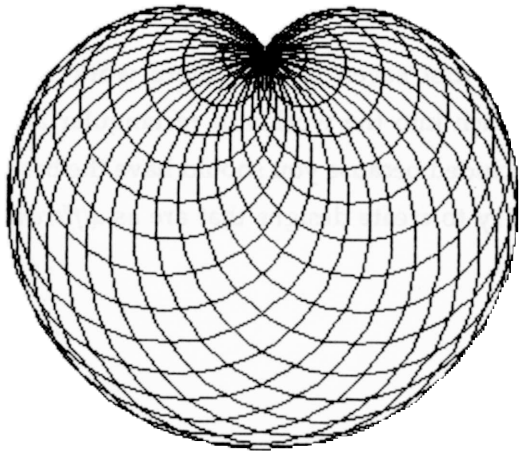
There are ten exercises this time. The first seven involve more work with circular grids, while the last three branch into the worlds of rectangular and random grids. There are fewer hints included because you should have the basic skills now to be your own designer. Try to create designs that are as *different* from the ones in this chapter as possible.

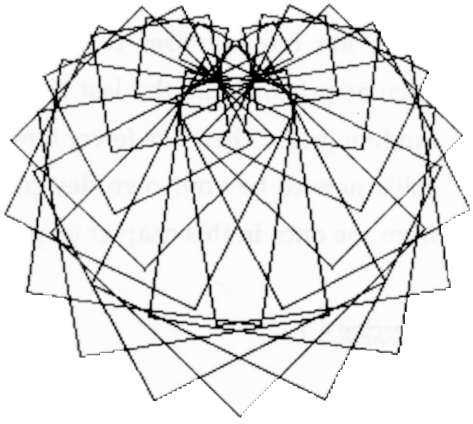
Exercise 4.1

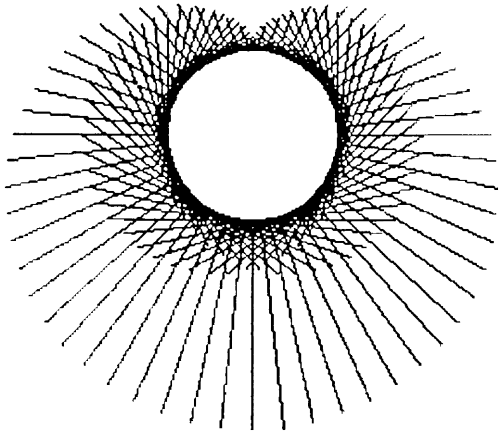
The nephroid on page 148 is composed of many individual circles. (Can you count the actual number of component circles from the design itself?) The method for locating and drawing each of these component circles was described on the last few pages of this chapter.

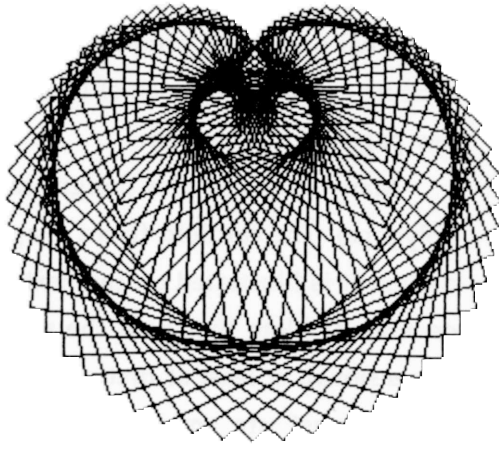
Many other complex designs can be built up entirely from circles (or other polygons). The cardioid (heart shape) is another example of such a composite design. It is illustrated on the next page. Can you suggest how the cardioid is generated? Note that both the nephroid and the cardioid seem to be three-dimensional, at least at first glance. Examination, however, will show that the "depth cues" are ambiguous and not totally consistent with "real" world perspective. This kind of ambiguity, though, can add interest to an otherwise straightforward, geometric design.

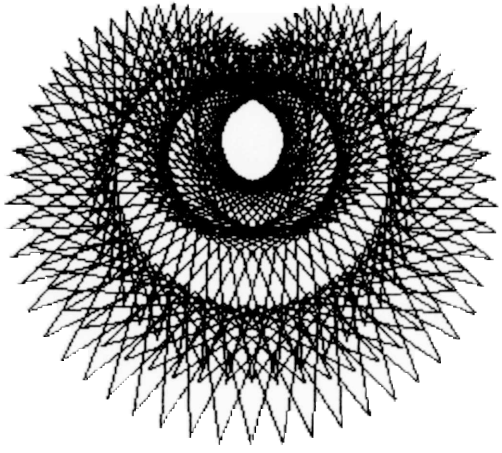
Cardioids





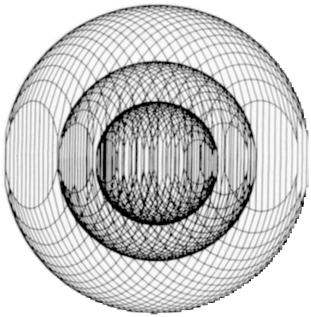


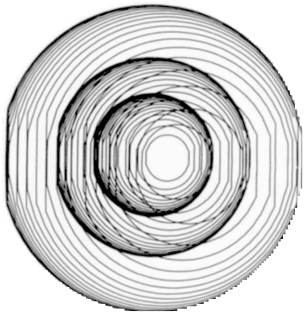


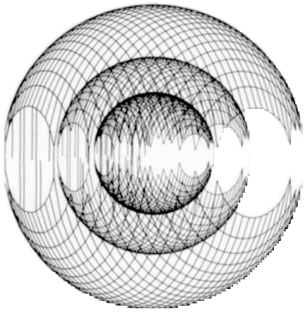


Now look at the following three designs. These seem to illustrate concentric spheres whose fronts are transparent but whose backs are opaque. What is going on here? Are the 3-D methods used to draw these globes similar to those used in the nephroid and cardioid designs?

Dream up some elegant Logo procedures to create designs inspired by nephroids, cardioids, and these transparent spheres.

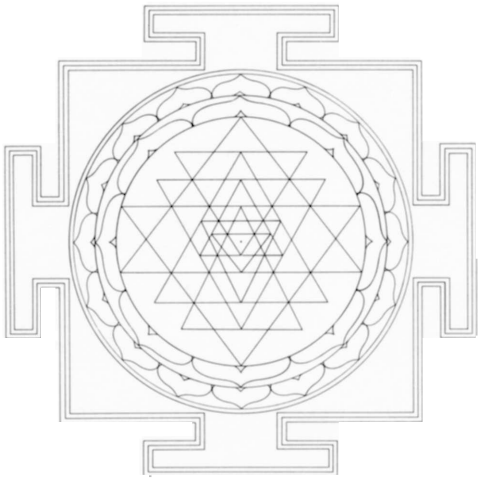


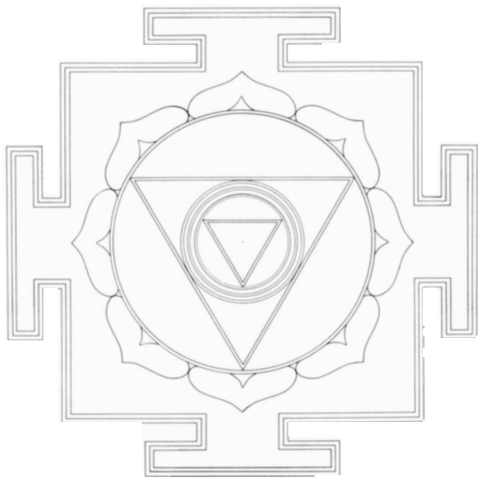




Exercise 4.2

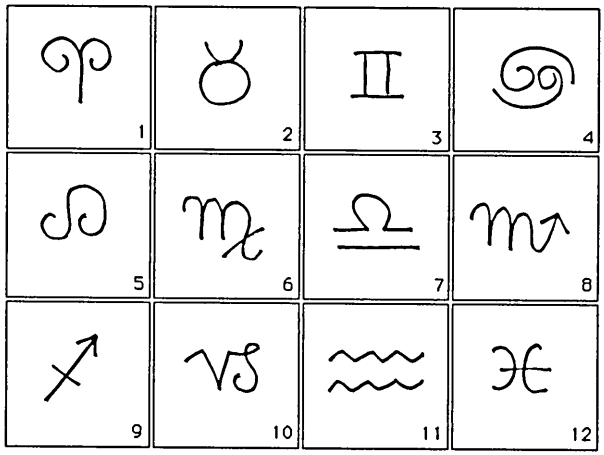
I once asked you to design a personal mark. Now we move from the individual to the cosmos. In Hinduism and Buddhism diagrams called *mandalas* are used to illustrate the structure of the universe and the place of the observer within it. They are also used in meditation and visualization exercises. Two examples are shown below. I would like you to generate a series of mandalas, each based on aspects of your *own personal cosmology*. Be ready to "explain" your diagrams.





Exercise 4.3

While we are thinking about the cosmos, take a look at the following. Here are the standard glyphs for the twelve signs of the zodiac, drawn by me. The series begins at (1) Aries and runs through (2) Taurus, (3) Gemini, (4) Cancer, (5) Leo, (6) Virgo, (7) Libra, (8) Scorpio, (9) Sagittarius, (10) Capricorn, (11) Aquarius, and (12) Pisces. Can you redesign each of them into the form of a circular grid? Simplify each symbol to the point beyond which it would lose its visual identity. You may want to borrow a book on astrology to read the traditional meanings of the signs so that you can incorporate them into your designs.



Exercise 4.4

Playing cards offer another set of designs that can be structured as circular grids. You might be intrigued to redesign the standard card suits: diamonds, clubs, hearts, and spades. Or maybe the face cards of King, Queen, Jack, and Joker. But these images may not be as challenging for you as the face cards of the tarot deck. (You should be able to borrow a deck of Tarot cards from the same person who lent you the astrology book for Exercise 4.3.) Listen to these wonderful face

card names: Magician, High Priestess, Empress, Emperor, Hierophant, Lovers, Chariot, Strength, Hermit, Wheel of Fortune, Justice, Hanged Man, Death, Temperance, Devil, Tower, Star, Moon, Sun, Last Judgment, World, and finally, The Fool.

Can you design a circular grid icon for one or more of these cards? Keep each of your designs as simple as possible.

Exercise 4.5

The standard clock face is a circular grid. Redesign it with symbols rather than numbers. Remember to keep track of seconds, minutes, and hours. But you may want to change the whole system of reckoning time. . .

Exercise 4.6

Design a generalized Gothic stone mason mark that has three arguments. One of these arguments should control the size of the design and the other two should control the shape. Don't limit yourself to the examples already shown; produce a series of designs that could be used for late-twentieth-century cathedral building.

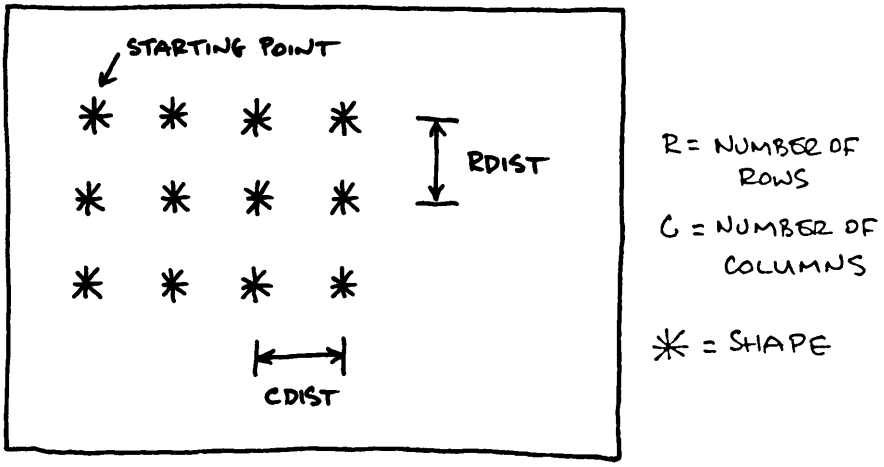
Exercise 4.7

We didn't spend much time on fractals in this chapter. Go back and look at the FRACTALGONs. Can you dream up some fractals of your own? We will look at fractals again in the exercise section of Chapter 7.

Exercise 4.8

Design a Logo procedure that will put a shape into a rectangular grid. You should be able to define the number of rows and columns that the procedure will use and the distances between them. You might want to be able to change the shape that goes into this grid as well. Make sure your rectangular grid can be placed at different starting places on the screen.

Remember: the secret to success is always to go through the sketch/turtle walk/more sketches scenario. Here is a little design that will get you started on your walk. Break up the tasks into sections, write a Logo procedure for each, and put the sections together into a master procedure that controls the action.



Exercise 4.9

Design a procedure that will place a certain number of STARGONs randomly on the screen (a random rectangular grid). Look up the RANDOM command in your Logo language manual and carry out some visual experiments with it. Try to say something about how you feel when you look at the "designs" produced by your random machine.

Exercise 4.10

Piet Mondrian was a painter who specialized in rectangular grids. Find one or two postcards, or small reproductions, of Mondrian's work that you like. The portability is important because I want you to pin them up over your desk or work space. Now, spend a lot of time looking at these Mondrians. Can you characterize, very roughly, the grid themes of your Mondrian examples?

Can you translate your characterization of Mondrian grid themes into a Mondrian model?

The end of the chapter at last

Sometimes people think that Logo procedures must be complicated to be good. Wrong. Go back and look at NEPHROID; it really has only one line. Here are two *rectangular* grid procedures that are short and surprising. They should get you into shape for Chapter 5.

```
TO ZIP :SIZE :GROWTH
 ; Set your pen to the reversing color,
 ; and try ZIP 10 5.
 REPEAT 2 [ FD :SIZE RT 90 ]
 ZIP :SIZE+:GROWTH :GROWTH
END

TO ZIPM :SIZE :GROWTH
 ; Set your pen to the reversing color,
 ; and try ZIPM 10 1.05.
 REPEAT 2 [ FD :SIZE RT 90 ]
 ZIP :SIZE*:GROWTH :GROWTH
END
```

Chapter 5
Rectangular and Random Grids

"Teased by the crisscross of the world."
John Galsworthy

Placement, motif, and symmetry

By now you should have an appreciation for the power of visual models to aid your thinking about shapes. Simulating stone mason marks encouraged you to think about clusters of shapes and their relative placement. In these mason mark designs, placement was carried out in reference to a single point that became the center of the final composition. Often, an arrangement of objects around the center was repeated to create a special kind of balanced figure, a symmetric one. But what is symmetry?

Let's be as simple as possible about this notion. To say that a design is symmetric means that each part of the design has the same organization as some other part. It is the repetition of parts that creates a certain balanced quality in the whole. All balanced designs, however, are not necessarily symmetric. Look back at the Delaunay machine in Chapter 3 for an example of balance without symmetry. Can you imagine a design that is partially symmetric or partially not?

Specifically, symmetry describes a correspondence in size, shape, and relative position of parts in reference to a dividing line, or median plane, or

about a center or axis.

Circular symmetry is a special form of design balance in which the correspondence of parts is about a central point. Look back at the stone mason marks you produced in Chapter 4. How would you describe the nature of the circular symmetry exhibited in them? Look, too, at the designs you made to represent your own character in the personal mark excursion, Exercise 3.6. How much symmetry do you find there? We know something of balanced personalities, but what would a symmetric character be? Put your comments in your notebook.

This chapter will investigate another form of balance—rectangular symmetry. Here the correspondence of elements will be relative to the rows and columns of a rectangular grid rather than to a central point.

Let me have one last word about symmetry before we move on to modeling it explicitly. Symmetric designs can be viewed on two levels. We can divide the composite design into the *motif* and the *rule of repetition*. For example, look back at two symmetric images in the last chapter—STARGONs and NEPHROIDs. The STARGON motif is the kinked line and the repetition rule is the machinery of CNGON. In the nephroids, the motif is the circle and the repetition rule is stated as the last line of the NEPHROID procedure (see Chapter 4).

A small review of list mechanics

The first example of rectangular symmetry will be taken from Exercise 4.8. This problem asked you to design a generalized rectangular-grid machine. But first we will need to review some Logo list-handling mechanics in preparation for the approach I would like to take to this problem. Please read the list section in your Logo manual, and I'll quickly go over the most important list operations.

List definition

Lists are collections of Logo elements enclosed by square brackets [ ]. Be careful not to confuse these square brackets with two other kinds of brackets available on most computers: round brackets ( ) and curly brackets { }. Round brackets are usually called parentheses.

Here are some examples of lists that contain one or more elements that are not themselves lists:

```
[JIM]
[JIM CLAYSON]
[100 200 3 -20]
[JIM LOOKS 85 BUT ACTS 16]
[STARGON 4 40 .5]
[FD 100 RT 45 FD 100]
```

Lists may contain elements that are themselves lists. That sounds a bit recursive, doesn't it?

```
[REPEAT 4 [FD 25 RT 90]]
[[[MIT] [CHICAGO] [LONDON]] [[AMERICAN COLLEGE IN PARIS]
 [PARSONS]]]
```

Lists may also contain no elements. These "empty lists" are very special beasts, and we may find them useful later on:

```
[ ]
```

The following list is not empty. It has two elements, each of which is an empty list:

```
[[ ]  ]]
```

Values of variables can be lists

The value of a variable can be a list as well as a number or a word.

```
MAKE "JIM  21
 ; Variable's value is a number.
MAKE "JIM  "HAPPY
 ; Variable's value is a word.
MAKE "JIM  [STARGON 5 50 .5]
 ; Variable's values is a list.
```

The READLIST construct

Let's design a procedure to carry out the following tasks:

Ask the user to type a special x-y screen location on the keyboard.
Put the two numbers typed into a list.
Make this list be the value of a global variable.

```
TO GIVE.PT
 ; To create a global variable named
 ; :POINT whose value will be a list
 ; containing the x and y values typed.
 PRINT [Give the x and y values of the point]]
 MAKE "POINT READLIST
END
```

Notice that the PRINT command can be used to print a message on the screen. The message to be printed must be a list, but the outer brackets will not be printed.

Type GIVE.PT and it will talk to you by printing the message: "Give the x and y values of the point."

You will notice another new command, READLIST, on the line after the PRINT statement. When Logo arrives at a line that contains a READLIST command, Logo will wait until something is typed on the keyboard. Whatever

is typed is placed within a single list, and this list is made available at the location of the word READLIST.

The line MAKE "POINT READLIST, causes the following three operations to be carried out:

1. Logo waits until something is typed (for example, 100 200).
2. Logo puts whatever has been typed into a list (for example, [100 200]).
3. Logo creates a global variable named :POINT and makes its value equal to the list made available by READLIST (for example, typing :POINT would now give [100 200]).

Try the procedure. Type GIVE.PT, type in two numbers, and check to see if a variable called :POINT has been created and given the values you typed.

Taking individual elements out of a list

Now, let's send the turtle to the screen address that is now stored within the list named :POINT. You might think that the following should work:

```
SETXY :POINT
```

No. We have to take the numbers out of the list before the SETXY command is happy. SETXY needs two arguments, not a single list that has two numbers inside it.

There are many list manipulation commands in Logo, and you should review them in your own language manual. We will discuss only a few of these commands here. FIRST list returns the first element of that list; LAST list returns the last element; BUTFIRST list returns the list with its first element gone; and BUTLAST list returns the list with its last element missing. Some quick examples:

```
FIRST [A B C]  returns A
LAST [A B C]  returns C
BUTFIRST [A B C]  returns [B C]
BUTLAST [A B C]  returns [A B]
```

These commands can be used in combination:

```
FIRST BUTFIRST [A B C]  returns B
LAST BUTLAST [A B C]  returns B
```

Two final commands before we send the turtle off to :POINT. Sometimes you don't know how many elements there are in a list named :JIM. COUNT :JIM returns the number of elements in the list.

Finally, suppose you have a list with many elements and you want the fifth one. You could say: FIRST BF BF BF BF :JIM. But there is a more concise command: ITEM 5 :JIM returns the fifth element of :JIM. Be careful you don't do something like this, though: ITEM 10 [PARIS]. Why?

So let's send the turtle off to that location stored as the list :POINT.

```
TO GO.PT
 ;  To send the turtle to the x-y position
 ;  stored as the list value of :POINT.
 PU
 SETXY FIRST :POINT LAST :POINT
 PD
END
```

Putting several Logo elements into a list

A single example will make the point.

```
SENTENCE [A B] [N M]  returns [A B N M]
SENTENCE  100  -50  returns [100  -50]
```

Look up SENTENCE in your manual and compare it with the command LIST. Carry out a few experiments to see for yourself how they are different.

While you have the manual in front of you, take a peek at LPUT and FPUT. How are they different from SENTENCE and LIST? These list manipulators may seem a bit ponderous at first, but you will soon get a firm, visual sense of how they work. And that will help you decide when they are useful.

Regular rectangular grids

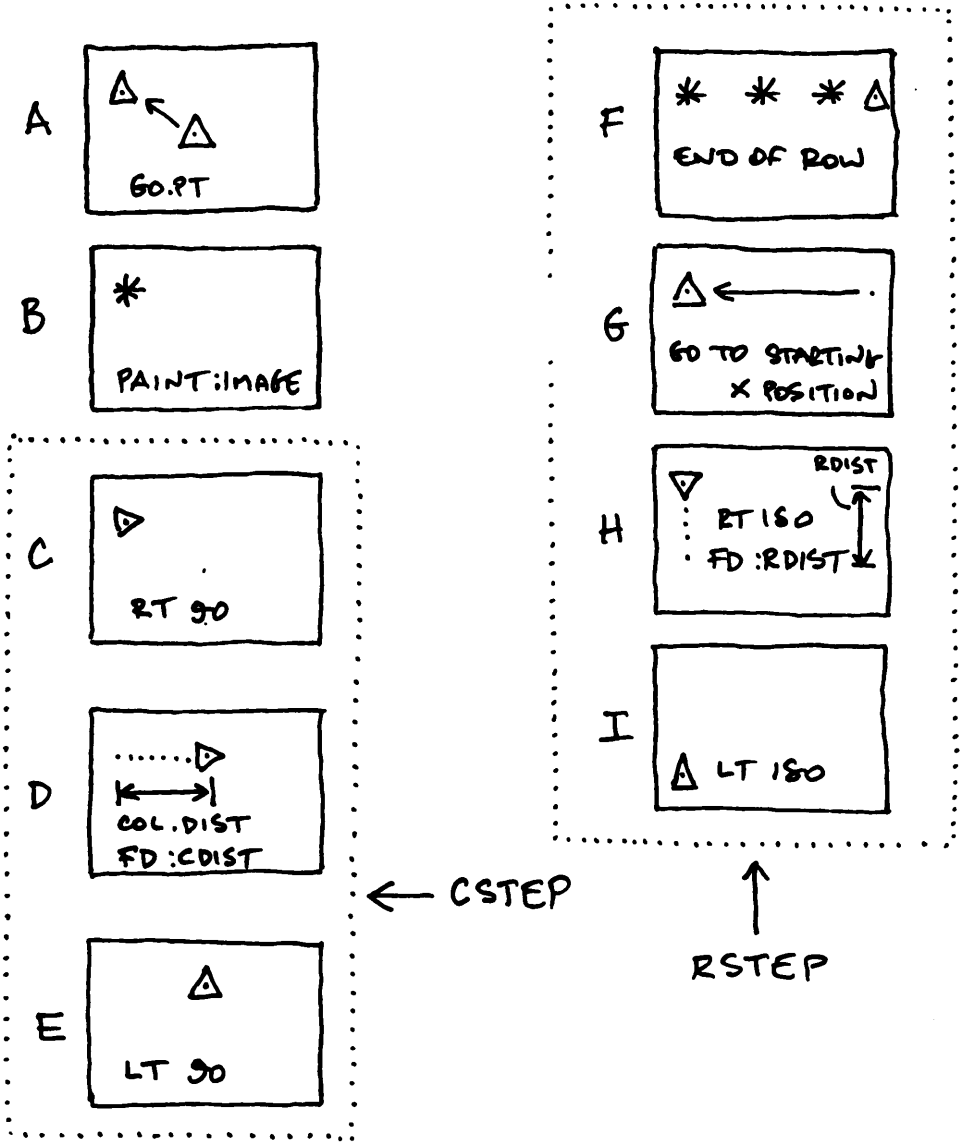
Now we can get on with Exercise 4.8. "Design a Logo procedure that will put a shape into a rectangular grid. You should be able to define the number of rows and columns that the procedure will use and the distances between them. You might want to be able to change the shape that goes into the grid as well. Make sure your rectangular grid can be placed at different starting places on the screen."

Turtle-walk story boarding

As usual, the best way to approach a complicated Logo problem is to do a turtle-walk diagram. A turtle-walk diagram is similar to the story boards used to plan TV and cinema advertisements. The story board is a series of small, sketched cartoons that illustrate the major events to be covered in the film. Each individual cartoon is accompanied by a few written comments. Sounds like what we have done with turtle walks: sketch the important "events" and comment on them in words.

On the next page is an example of how an illustration and design student used the story board approach to structure the rectangular grid exercise.

Story board for the rectangular grid



Descriptions of the story board cartoon screens

"*Cartoon A* indicates the preparation that must be carried out before the grid is started. The screen must be cleared and the `GET.PT` procedure run to define the upper-left-hand-corner starting point of the grid. Somehow, the shape notation of the little figure that will make up the grid needs to be done here, too. I don't yet know how to do that, though, so I'll just assume that something, maybe a variable called `:MOTIF`, 'holds' the necessary shape notation. `GO.PT` puts the turtle at the first grid position.

"*Cartoon B* shows the drawing of one individual figure at any point in the grid. I'll assume that this can be done by using whatever is inside `:MOTIF`. My notation will be: `PAINT :MOTIF`. (Still don't know how.) `PAINT` will take the turtle from its current grid location, draw the little shape inside `:MOTIF`, and then return the turtle to the grid location on which `PAINT` found it. So some kind of *state transparency* must be respected by `PAINT`.

"*Cartoons C, D, and E* sketch the movement of the turtle from one column to the next. The turtle must move over to the next column until it has completed all the columns of any one row. To do this, the turtle needs to pick up its pen, turn right 90, move forward the between-columns distance, turn left 90, and put down its pen. I'll call this procedure `CSTEP`, for column stepping. `CSTEP`'s single argument will be the distance between columns.

"*Cartoons F, G, H, and I* sketch the movement of the turtle from the end of a finished row to the beginning of the next row, if an additional row is required. Cartoon G shows the turtle, after lifting its pen, moving left to the x value of the starting point, `:POINT`. The turtle then turns right 180 and goes forward by an amount equal to the between-row distance. The turtle turns another right 180 and puts down its pen. The name of this procedure will be `RSTEP`, for row stepping. `RSTEP`'s single argument will be the distance between rows."

Story board cartoons into Logo notation

Everything that the illustration student said above is correct. Her intuition was very good and her idea about PAINT :MOTIF (before she knew about lists) is very elegant. Let's use her story board "characters" and translate them into formal Logo notation.

The procedure to set up the upper-left-hand starting point has already been written. It's GIVE.PT. Our procedure GO.PT sends the turtle to the starting place.

What about PAINT :MOTIF? What is :MOTIF?

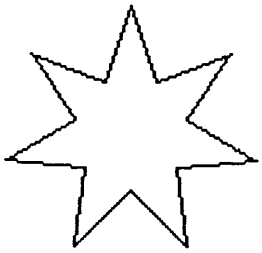
Let's think of :MOTIF as being a list of the Logo commands that will draw the needed figure. For example, suppose we want a grid made from individual stars with 7 points. We could make the value of :MOTIF be a list like [STARGON 7 10 .5]. How do we then "paint" this list?

Painting the :MOTIF list

We are already familiar with the REPEAT n [Logo commands] construct. This command could be used to get the MOTIF list painted—or drawn—on the screen. We could simply say: REPEAT 1 :MOTIF. A shorter way of saying REPEAT 1 [something] is RUN [something]. So we can answer the question of how to paint the list as follows: Make sure the turtle is where you want the motif to be, and then RUN the list that defines the motif.

Here is an example of RUNning a :MOTIF list:

```
MAKE "MOTIF [STARGON 7 10  .5]
RUN :MOTIF
```



Don't forget, though, that the MOTIF list must be state transparent. There is no problem about this with STARGON because this procedure always leaves the turtle where and how it found it.

Let's write a tiny procedure to ask that the elements of the :MOTIF list be typed on the keyboard, to place the results into list, and to make the list be the value of :MOTIF.

```
TO GIVE.MOTIF
 ; To set up the :MOTIF variable that will be
 ; used by the master grid-building procedure.
 PRINT [Give the Logo commands that define the shape]
 PRINT [The shape commands must be state transparent]
 MAKE "MOTIF  READLIST
END
```

To verify what we have done so far, type GIVE.MOTIF and then respond with a series of state-transparent Logo commands. Next, type RUN :MOTIF. Watch the turtle. Everything OK?

Let's name the master procedure that will put the pieces of the grid procedure together FLAG. FLAG will need four arguments:

```
:COLS     - number of columns in the grid
:ROWS     - number of rows in the grid
:CDIST    - distance between the grid's columns
:RDIST    - distance between the grid's rows
```

Before we write FLAG, we can write the much simpler CSTEP and RSTEP:

```
TO CSTEP :C
 ; To move the turtle over one column.
 ; The between-column distance is :C.
 PU RT 90
 FD :C
 LT 90 PD
END

TO RSTEP :R
 ; To move the turtle from the end of one row
 ; over and then down to the beginning of the next.
 ; The between-row distance is :R.
 PU SETX FIRST :POINT
 RT 180
 FD :R LT 180 PD
END
```

We can now easily fit the pieces together.

```
TO FLAG :COLS :ROWS :CDIST :RDIST
 ; To create a rectangular grid flag.
 ; :POINT holds the x-y starting position and
 ; :MOTIF holds a state-transparent shape.
 GO.PT
 RUN :MOTIF CSTEP :CDIST      <-- repeat this step
                                  for each of
                                  the columns

                                          <-- repeat these
                                              steps for
                                              each of the
 RSTEP:RDIST                                  rows
END
```

The circle notations used above are transposed into the REPEAT commands of the completed FLAG procedure below. Notice that one REPEAT is "nested" inside another REPEAT. This REPEAT nesting copies the circle nesting above.

```
TO FLAG :COLS :ROWS :CDIST :RDIST
 ; To create a rectangular grid flag.
 ; :POINT has the x-y starting position and
 ; :MOTIF has a state transparent shape in it.
 GO.PT
 REPEAT :ROWS [REPEAT :COLS [RUN :MOTIF CSTEP :CDIST] -
                            RSTEP :RDIST]
END
```

That's it. The compactness of the FLAG procedure is possible because we used lists for "holding" our MOTIF description, nested REPEAT commands, and short, support structures—GIVE.PT, GO.PT, GIVE.MOTIF, RSTEP, and CSTEP.

But don't overlook the difficulty of understanding that double repeat operation. Try getting an intuitive feel for it by running FLAG with STEP turned on; watch carefully what happens.

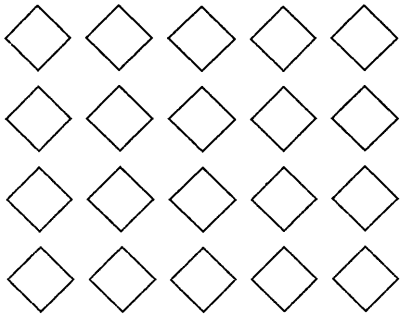
Did you rejoice that none of the procedures associated with FLAG used recursion? The REPEAT construct and recursion methods are often alternative ways of accomplishing a modeling task. The choice between the two approaches is often a personal one; one method may simply seem more aesthetically pleasing to you for a specific problem. Occasionally, you may feel strongly that recursion is the right visual metaphor for what you are modeling and other times that it is not. Exercise 5.2, at the end of this chapter, asks you to restructure FLAG using recursion, avoiding all uses of REPEAT. You should then be able to make your own comparisons between these alternative styles.

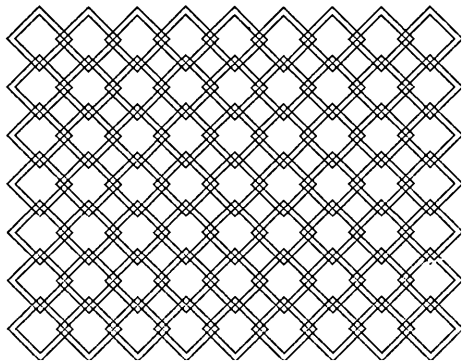
On the next page you will see an exploration of grids made up only of squares. Can you guess :MOTIF's value for each of the illustrations?

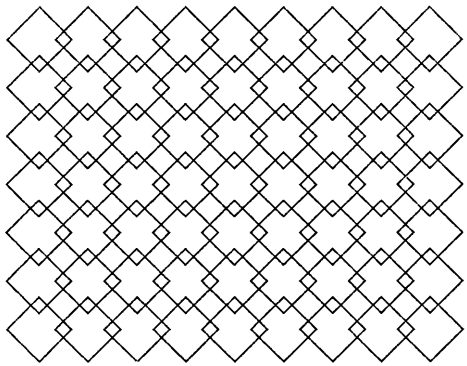
State transparency in the MOTIF list

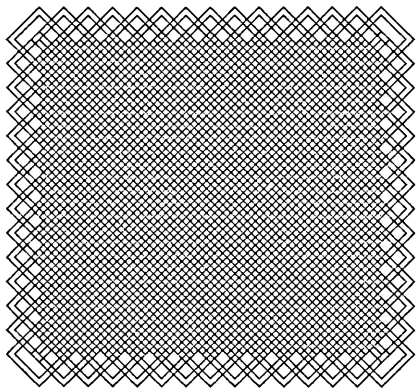
The FLAG-produced grids on the previous page all used either CNGON or CONGON in their MOTIF lists. Remember how careful we were to design those two procedures so that they both left the turtle in the same position in which it was found? This is state transparency. If you include procedures in a FLAG MOTIF list

Rectangular grids composed of squares

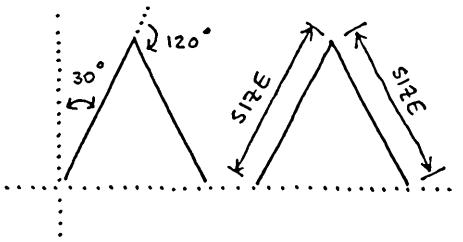








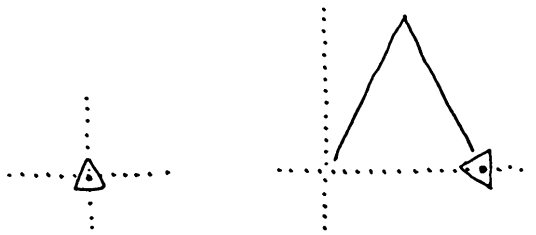
that are already state transparent, everything is fine. If, on the other hand, you wish to make a grid design from a non-state-transparent procedure you have some additional work to do. Let's review these ideas by going through the exercise of turning a non-state-transparent procedure into a state-transparent one and then making grids with it. Suppose we want to make a grid with a teepee-shaped figure like the following:



Here is a procedure for drawing the teepee shape:

```
TO TEEPEE :SIZE
 RT 30 FD :SIZE
 RT 120 FD :SIZE
END
```

This procedure is not state transparent because it does not return the turtle to its original position. The starting and ending positions of the turtle are shown below.



There are two simple ways to make TEEPEE state transparent. The first is the most obvious: simply move the turtle backward to where it began. Call this procedure TP1.

```
TO TP1 :SIZE
 ; First approach to making TP state transparent.
 RT 30 FD :SIZE
 RT 120 FD :SIZE
 BK :SIZE LT 120
 BK :SIZE LT 30
END
```

This method doubles the length of the procedure, and the additional length obscures the procedure's design. Procedures should be as short and as descriptive as possible; that's good modeling style. Perhaps there is another way to make the state transparency of the procedure more obvious.

We really want to say something quite simple to the turtle: "Remember where you started from—and in which direction you were heading—and then go back to that position after you have finished drawing the teepee." The second approach does this explicitly; but we need two new supporting procedures to make it work nicely.

The procedure RECORD.POS assembles the turtle's current x-y position and heading into a three-element list called :POS. The operation of RESTORE.POS should be obvious.

```
TO RECORD.POS
 ; Records the turtle's current position state.
  MAKE "POS ( SE XCOR YCOR HEADING )
END
```

Note: when the command SE has more than two arguments, as in the situation above, parentheses must be placed before SE and after a space left behind the last argument. Verify this in your own Logo manual.

```
TO RESTORE.POS
 ; To restore the turtle to the position state
 ; recorded in :POS.
 PU SETXY FIRST :POS FIRST BF :POS
 SETHEADING LAST :POS
 PD
END
```

Here, then, is an *obviously state-transparent* teepee maker:

```
TO TP2 :SIZE
 RECORD.POS
 TEEPEE :SIZE
 RESTORE.POS
END
```

You will find many more uses for RECORD.POS and RESTORE.POS.

A Mayan-inspired grid that vibrates

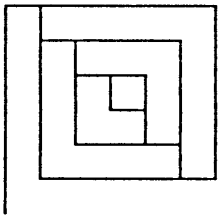
I hope the square and teepee grids make you think about rectangular symmetries and that you are now impressed with the help Logo models give us in exploring visual ideas. I will admit, though, that these grids are not the most exciting images to look at. They lack surprise and "quirkiness." They are, in fact, too balanced and too symmetric. So I am pleased to present a student-designed grid that exhibits an odd, surprising quality—within a symmetric framework. I'll let the student describe his own work.

"I wanted to make a grid from square-ish spirals that wrapped back on themselves. I needed to use the RECORD.POS and RESTORE.POS procedures because I wasn't sure where my spiral would end up, and I had to make them state transparent. I came up with the Mayan description of the grid only after I looked at my results. I was surprised about the vibrating quality of the grid; I tried to explore the source of the vibration by changing the arguments that I gave to my procedures."

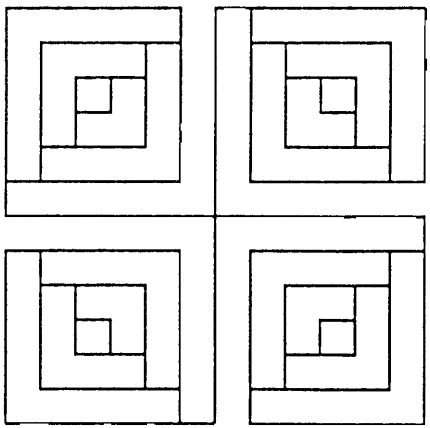
Here are the two procedures the student needed for his :MOTIF list:

```
TO SPIRAL :LEG :FAC :LEV
 IF :LEV < 1 [STOP]
 REPEAT 2 [FD :LEG RT 90]
 SPIRAL (:LEG-:FAC) :FAC (:LEV-1)
END

TO LAY :LEG :FAC :LEV
 ; Position 4 spirals at 90 degree
 ; intervals around a central point.
REPEAT 4  [RECORD.POS -
           SPIRAL :LEG :FAC :LEV
           RESTORE.POS RT 90]
END
```

```
SPIRAL 90 15 12
```

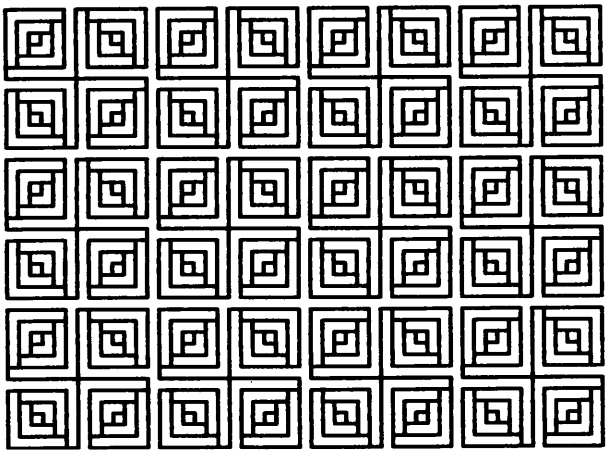


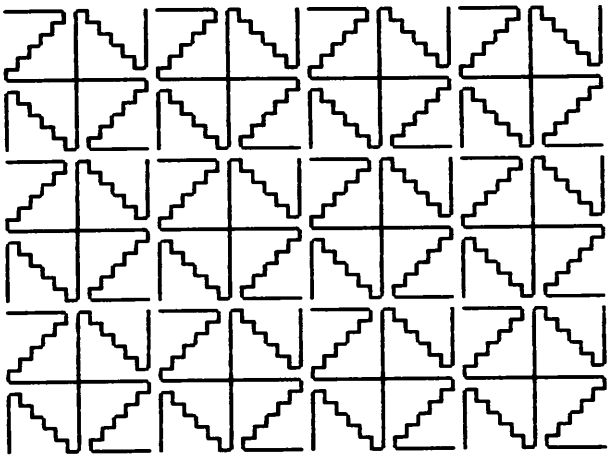
```
LAY 90 15 12
```



Two Mayan grids

For both illustrations below, the `:MOTIF` list was `[LAY 30 5 12]`. The `FLAG` arguments were `4 3 65 65`. The reversing pen color was used for the second illustration.





Adding random components to rectangular grids

One effective way to make rectangular designs more visually entertaining is to add limited amounts of randomness to them. I would like to introduce this idea by exploring Exercise 4.9: "Design a procedure that will place a certain number of STARGONs randomly on the screen (a random rectangular grid)."

How did you approach this? No doubt you looked up the RANDOM command in your Logo manual and found that it could produce "random numbers"; but what are those? The exercise of generating random grids will give you a visual model of what these random things actually look like. You need a good feel for these random numbers because we will be using the idea of randomness for the rest of this chapter and in several future chapters.

Random numbers

The Logo command RANDOM requires a single, integer argument. Logo then produces an integer in the range from 0 to the argument minus 1. For example, RANDOM 4 would produce a single number that could be 0, 1, 2, or 3. Each of these numbers is *equally likely* to be returned by Logo, and it is very unlikely that you could correctly guess what Logo would respond every time you type RANDOM arg.

Guessing a single RANDOM 4 outcome correctly is tough. It is even tougher to predict correctly the *pattern of the numbers* given by Logo if you should type RANDOM 4 over and over again. There may indeed be a pattern, but it cannot be guessed easily before it occurs. Unless, of course, you are exceptionally lucky, have ESP, or know the model that your version of Logo uses to generate random numbers.

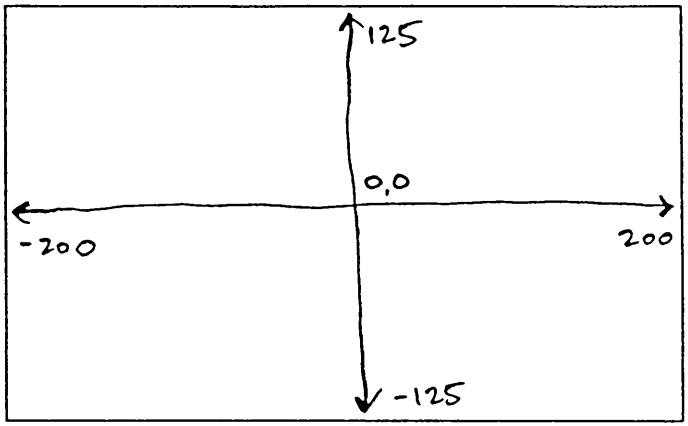
Let's design a Logo procedure to generate a series of random numbers so that we can look at them. Here is an example of such an exploratory tool.

```
TO RANDOM.NUMBER :RANGE :N
 ; To PRINT :N random numbers in the range 0 to :RANGE.
 IF :N < 1 [STOP]
 PRINT RANDOM (:RANGE+1)
 RANDOM.NUMBER :RANGE (:N-1)
END
```

Note several things about this procedure. First, the use of PRINT to display on the screen the value of a calculation. Other examples of this would be PRINT SIN 43 or PRINT variable. Second, remember that RANDOM arg outputs a random number that can never be larger than arg - 1. If you want it to be possible for the random number to be as large as, say, :RANGE, you will have to use RANDOM (:RANGE+1).

Random screen locations

We next need to decide how the RANDOM function can be used to generate not just random numbers but random x-y screen positions on your screen. Obviously we need two random numbers for each position, and the range of these numbers must correspond to the numbering convention of the screen. Because different screens have different maximum dimensions, let me show you the screen size that I will be using. You may have to make some adjustments:



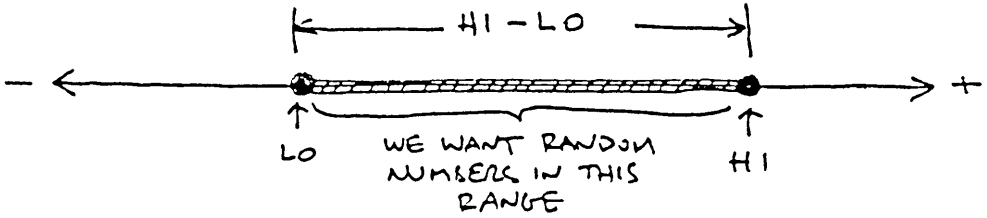
Notice that the x coordinates of my screen run from -200 to +200, and the y coordinates run from -125 to +125. To pick a random position on this screen requires negative as well as positive random numbers. But since RANDOM will not produce negative numbers, we must be a bit clever.

Generalizing RANDOM

What we need here is a Logo procedure that will produce a random number within a defined range that may include negative as well as positive integers. Let's design a new procedure that improves upon and extends RANDOM to do this.

Call the new procedure RR for random numbers within a defined range of possible numbers. RR will need two arguments, the first to define the lower end of this range and the second to define the higher end. Call them :LO and :HI. But how do we do it?

Perhaps a drawing might help. We want random numbers that are never less than :LO and never higher than :HI. We want these numbers here:



We could express this a little differently by saying that we want numbers no less than :LO and no more than :LO plus the difference between :HI and :LO. Perhaps we could generate random numbers between 0 and the difference between :HI and :LO and then add these to :LO.

We already know how to generate random numbers between 0 and the difference between :HI and :LO using RANDOM:

```
RANDOM 1 + (:HI - :LO)
```

This expression will produce numbers never smaller than 0 and never bigger than (:HI - :LO). The random number we want, the numbers that fall in the range between :LO and :HI, could be found with:

```
:LO + RANDOM 1 + (:HI - :LO)
```

We thus have the idea we need to write the procedure:

```
TO RR :LO :HI
 ; To generate a single random number
 ; in the range defined by :LO and :HI.
 OP (:LO + RANDOM 1 + (:HI - :LO))
END
```

Note the appearance of the new command OP, short for OUTPUT. Look it up in your Logo manual. Why do we need OP in the procedure RR?

Experiment with RR. Notice that it works fine for negative numbers. For example:

```
RR  -100  -50
RR  -100  50
```

But what's wrong with the following?

```
RR 100  -100
```

Using RR to generate random screen locations

We can easily generate a random screen location for my screen with the following:

```
SETXY (RR -200 200) (RR -125 125)
```

Let's use this SET idea to send the turtle to a random position on my screen.

```
TO GO.RANDOM.SCREEN
 ;  To position the turtle at a random x-y screen point
 PU SETXY (RR -200 200) (RR -125 125) PD
END
```

Putting the random grid procedure together

We can now put the pieces together. GO.RANDOM.SCREEN sends the turtle to a random screen position. Next, we want a :MOTIF placed at this point. Then we send the turtle on to the next random position with GO.RANDOM.SCREEN. The :MOTIF to be drawn at each stopping position can be defined, just as we did in the rectangular grid exercise, as a list of Logo commands. We call the master procedure that combines the pieces RANDOM.PLACER.

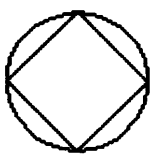
Let's build a few :MOTIF lists to test out RANDOM.PLACER.

```
MAKE "ORNAMENT1 [STARGON 5 20 .5]
MAKE "ORNAMENT2 [CNGON 4  30 CNGON 30 30]
```

Now we could tell Logo to RUN :ORNAMENT1 or RUN :ORNAMENT2. Remember that this is equivalent to typing:

```
RUN  [STARGON 5 20 .5]
RUN  [CNGON 4  30 CNGON 30 30]
```

Here is what these two :MOTIF lists look like:



Naming is power

Much of Logo's power comes from its ability to make a name, like `:ORNAMENT1`, have the value of—or be equivalent to—something else that is much more complicated. This super shorthand is enormously useful. Once we decide how to do something in Logo, we give it a name. Then all we need to remember is the *name of the things to be done* and not the complicated bits and pieces actually required to get them done.

Building the visual model of random numbers

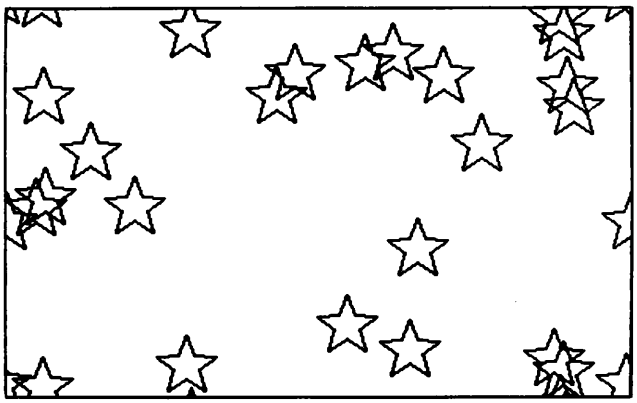
No more talk; here it is.

```
TO RANDOM.PLACER  :MOTIF  :T
 ; To place the shape list given as the argument
 ; :MOTIF randomly on the screen  :T times.
 IF :T < 1 [STOP]
 GO.RANDOM.SCREEN
 RUN :MOTIF
 RANDOM.PLACER :MOTIF (:T-1)
END
```

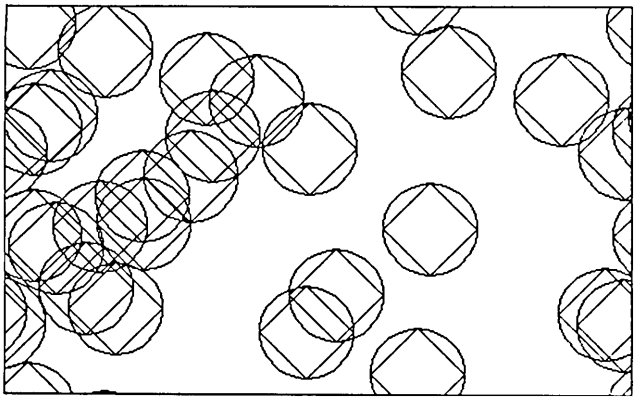
Visual experimentation

Try running this routine by typing the following:

```
RANDOM.PLACER :ORNAMENT1 25
```



```
RANDOM.PLACER :ORNAMENT2 25
```



Generalizing the random placer machine

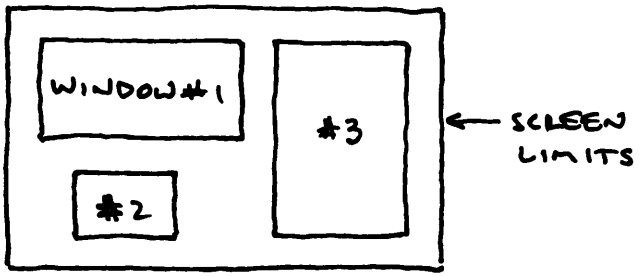
I was surprised by what I saw when I first looked at a random grid design. I expected to see a tangled "mess" of little :MOTIFs. Instead I found collections of figures that were pleasing and distinctive. The character of one random grid was often quite different from the character of another. The placement of the clusters of composite :MOTIFs produced a nice balance in some of the grids while unbalancing others. Some grids were lively, and others subdued. Many were neutral and boring.

I wondered if I could blend some of this randomness into the rectangular grids you have already seen. The addition of a little bit of randomness into a geometrical structure can be a way of giving life and animation to a static design.

But, before all that, let's clean up the random placer procedure and make it a little more general. First, an element of tidiness: I don't like the overlapping of :MOTIFs at the margins of the screen. If a figure is too near the bottom of the screen, for example, it wraps around and finishes at the top.

Second, I would like to be able to change the size of the "window" into which the figures are randomly placed. So far, we have been using the size of my entire screen as the window. If I could define the "window" size and define

its location on the screen, I could have several "fields" of randomly placed :MOTIFs on one screen. Here is a sketch of what I have in mind.



Because windowing might be useful when we attempt to add bits of randomness to structured patterns, let's start working on this window business first.

Defining a window as a list

Let's define the size and placement of a window in terms of a list. Arbitrarily, let's use the following convention: the first element in the list represents the y value of the top of the window, the second element is the x value of the right edge of the window, the third element is the y value of the bottom of the window, and the fourth element is the x value of the left edge of the window. I started at the top of the window and went around it in a clockwise direction. My screen would be represented as a window with the following characteristics:

```
MAKE "MY.SCREEN  [125  200  -125  -200]
```

Here is a procedure that will take a window list as its single argument and draw the window on the screen.

```
TO BOX :WINDOW
 ; To draw the outline of a rectangular
 ; window list on the screen.
 RECORD.POS
 PU SETXY (LAST :WINDOW) (FIRST :WINDOW) PD
 ; Positions turtle at top left-hand corner of window.
 SETX  (FIRST BF :WINDOW)    ; Draw top of window
 SETY  (LAST BL :WINDOW)     ; Draw right edge of window.
 SETX  (LAST :WINDOW )       ; Draw bottom of window.
 SETY  (FIRST :WINDOW)       ; Draw left edge of window.
 RESTORE.POS
END
```

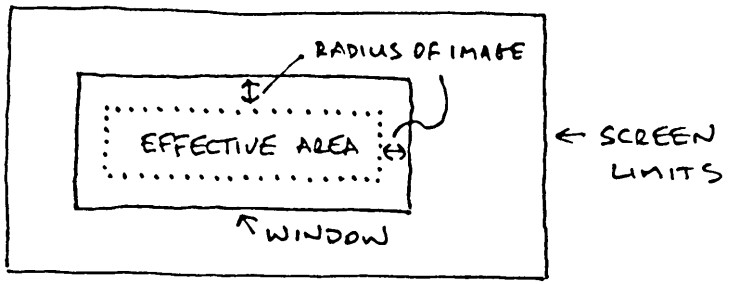
Note that BOX begins by positioning the turtle at the top left-hand corner of the window and then draws the window by moving the turtle around the window in a clockwise fashion. BOX is state transparent, too. BOX :MY.SCREEN will draw a nice frame around whatever images are on my screen.

Eliminating the overlap of :MOTIFs near the window edges

Suppose that the :MOTIF that we are using is:

```
MAKE  "PIP  [STARGON 5 20 .5]
```

If we were careful never to let :PIPs be drawn any closer than 20 units from each of the four window edges, we would never have any edge-overlap problem. We could just make the window smaller by 20 units in each of the four directions. A picture helps:

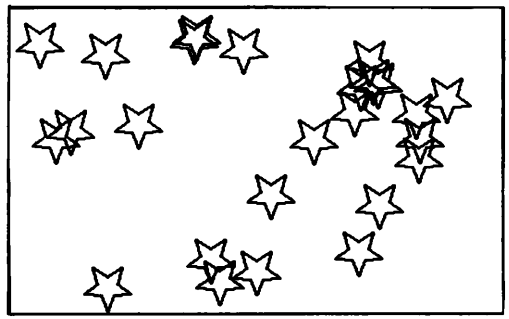


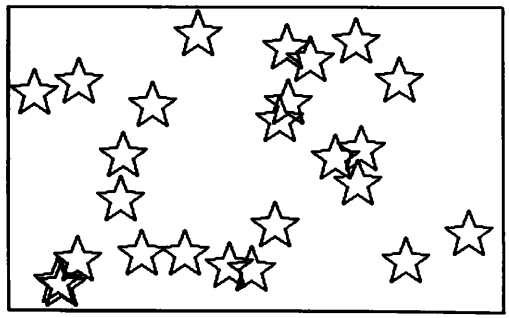
But how do we know how much to reduce the window when the :MOTIF list is changed to something else? If we knew the number of the element in the :MOTIF list that referred to the "radius" of the :MOTIF to be drawn, we could pluck out this element and use it in our calculations. We are now at a stage in Logo when Logo notation is easier to "talk" than English. So look at the following upgraded RANDOM.PLACER. I have called this new, extended version RANDOM.PLACER.X.

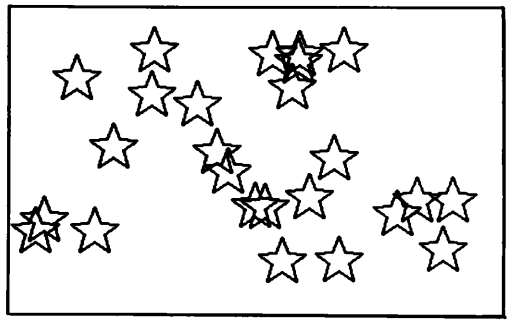
```
TO RANDOM.PLACER.X  :MOTIF  :WINDOW  :N  :T
 ; To randomly place :T :MOTIFs within a given
 ; :WINDOW with no overlaps at edges.
 ; :N is the number of the element within the :MOTIF list
 ; that defines the "radius" of the :MOTIF shape.
 (LOCAL "TOP "RIGHT "BOTTOM "LEFT)
 MAKE  "TOP    (ITEM 1 :WINDOW) - (ITEM :N :MOTIF)
 MAKE  "RIGHT  (ITEM 2 :WINDOW) - (ITEM :N :MOTIF)
 MAKE  "BOTTOM (ITEM 3 :WINDOW) + (ITEM :N :MOTIF)
 MAKE  "LEFT   (ITEM 4 :WINDOW) + (ITEM :N :MOTIF)
 REPEAT :T [PU -
            SETX RR :LEFT :RIGHT -
            SETY RR :BOTTOM :TOP -
            PD -
            RUN :MOTIF]
END
```

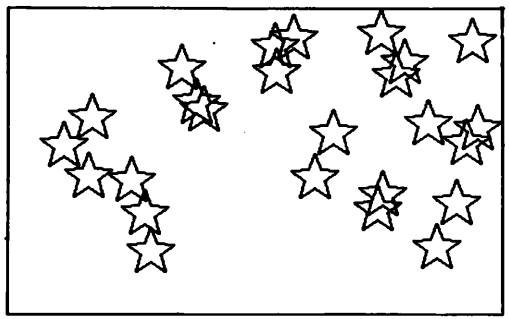
Note that the procedure GO.RANDOM.SCREEN has been incorporated into the body of RANDOM.PLACER.X and that a REPEAT statement has replaced recursion. I think the procedure "reads" well and needs no more explanation. So let's try it out.

Random grids with no edge overlap









Kandinsky grids

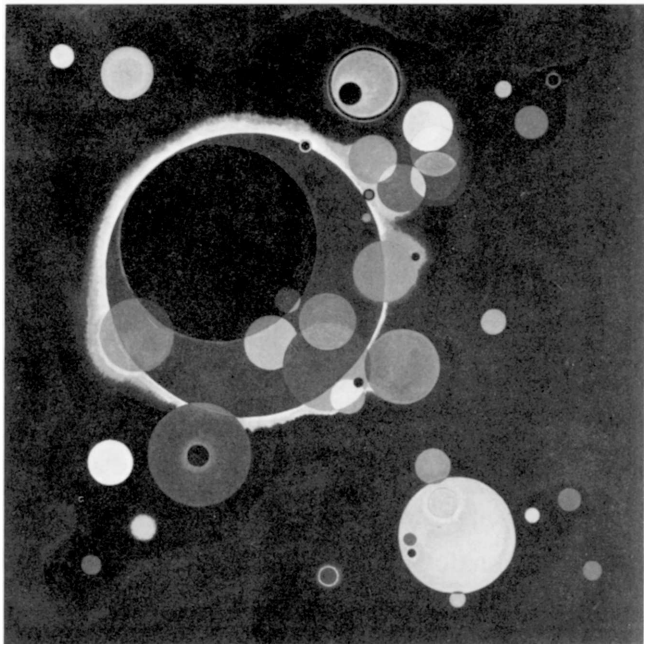
Wassily Kandinsky (1866-1944) was born in Moscow but trained in Munich. He totally abandoned realistic art for the abstract, a form he considered to be more spiritual. He made great use of simple geometric forms placed within frameworks that could be labeled "randomish."

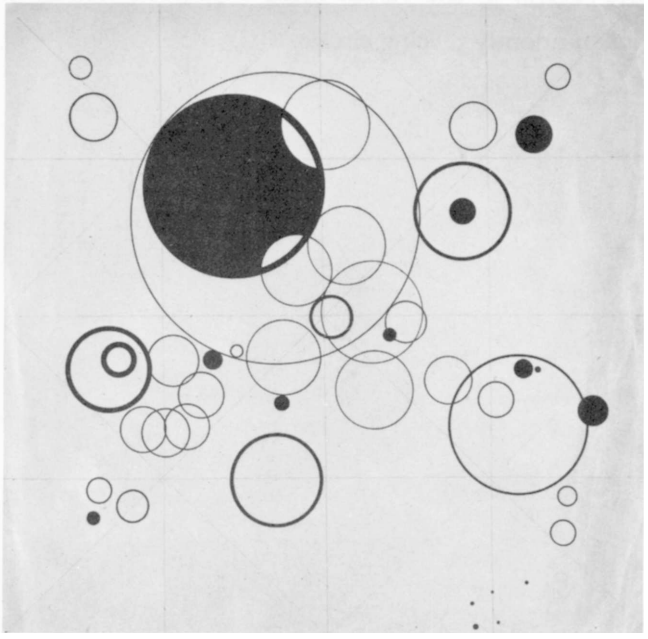
A good example of such work is "Several Circles, No 323," reproduced on the next page. The original painting is 1.4 meters square, and the circles glow with translucent light, not at all like this tiny and opaque reproduction. It is enormous and transparent. I have also included Kandinsky's pen-and-ink study for this painting.

Stare at this poor reproduction for some time or, far better, go out and find yourself some Kandinsky reproductions on postcards.

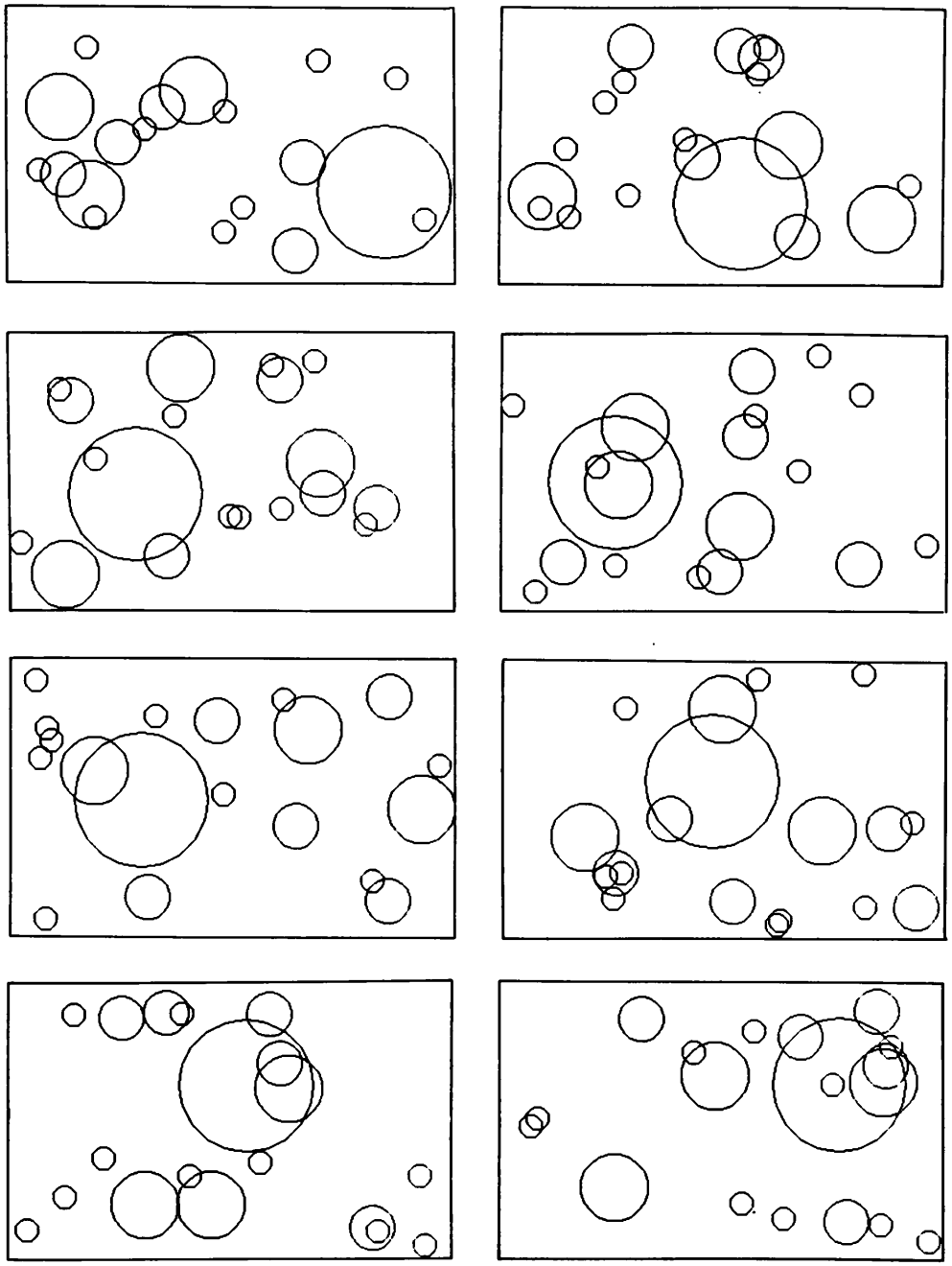
On the page following the reproductions, you will see a few of my Logo attempts to simulate Kandinsky's "Several Circles." They are, of course, pale imitations, but that is not the point. Now it's your turn. *Look* at this painting and try your best to simulate *several* of the visual ideas expressed within it. There is more going on than just randomly placing circles.

Kandinsky circles





Simulations of a Kandinsky painting



Random grids with multiple windows and multiple motifs

Look back at the random grids. There is no single focus of attention, no single place that immediately attracts the eye in any of these designs. A single focal point isn't always wanted in a design, but let's experiment with trying to place one within a randomized pattern.

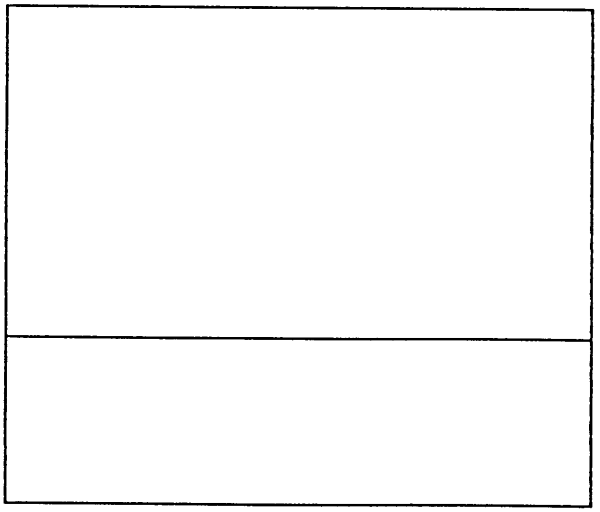
The following experiment is my own. You can certainly come up with different approaches, though your exercise should have the same goal as mine, namely, exploring the visual idea of *focusing random grids*.

I tried to focus a random grid by introducing two different sizes of the same shape (one large square versus many small ones) and by differences in placement (one large window for the small figures and one smaller window for the large figure). I divided my screen into two windows, one on top of the other.

```
MAKE "T.FRAME  [120  140  -40  -140]  ; The top window.
MAKE "B.FRAME  [-40  140  -120  -140] ; The bottom window.
```

Here is a picture of the two windows, using the BOX procedure.

```
BOX :T.FRAME
BOX :B.FRAME
```



Here are the two :MOTIFs that I will use:

```
MAKE "ORNAMENT3 [CONGON 4  5  1  4]
; A small square filled completely with black.

MAKE "ORNAMENT4 [CONGON  4  30  1  5]
; A larger square rimmed in black.
```

And the procedure that will carry out the experiments:

```
TO R.FOCUS.DEMO1
 CG RT 45
 RANDOM.PLACER.X :ORNAMENT3 :T.FRAME 3 20
 RANDOM.PLACER.X :ORNAMENT4 :B.FRAME 3 1
END
```

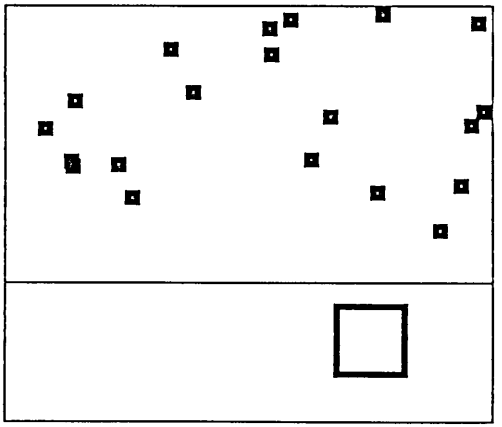
Here is my first experiment (the result is shown on the next page):

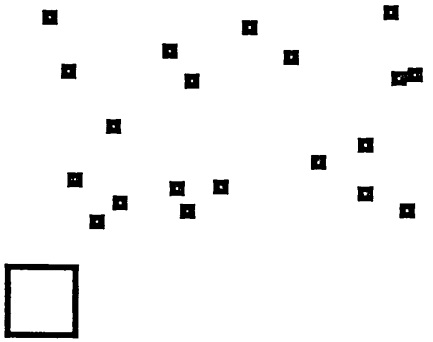
```
BOX :T.FRAME
BOX :B.FRAME
R.FOCUS.DEMO1
CG
R.FOCUS.DEMO1
CG
R.FOCUS.DEMO1
```

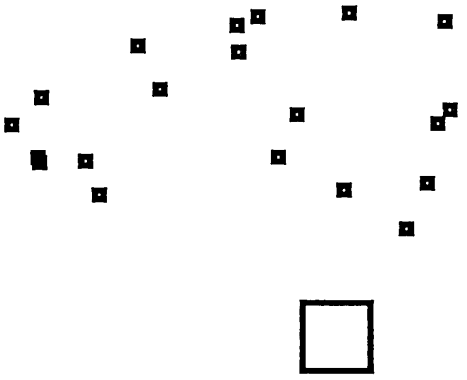
I then decided to look at the visual impact of a few modifications to this one focusing idea (two windows, two different sized :MOTIFs of the same shape). Here are my changes in the experiment. Notice that the two windows are now vertical rather than horizontal.

```
MAKE "L.FRAME [120  70  -120  -140]
MAKE "R.FRAME [120  140  -120  70]

BOX :L.FRAME
BOX :R.FRAME

TO R.FOCUS.DEMO2
 CG  RT 45
 RANDOM.PLACER.X :ORNAMENT3 :L.FRAME 3 20
 RANDOM.PLACER.X :ORNAMENT4 :R.FRAME 3 1
END
```

Focused random grids #1

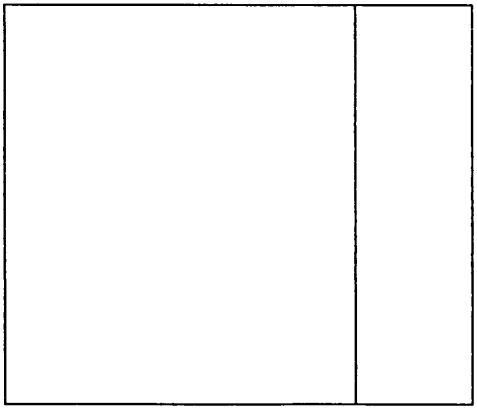






Here is my second experiment (the results are on the next page):

```
BOX :L.FRAME
BOX :R.FRAME
R.FOCUS.DEMO2
CG
R.FOCUS.DEMO2
CG
R.FOCUS.DEMO2
```



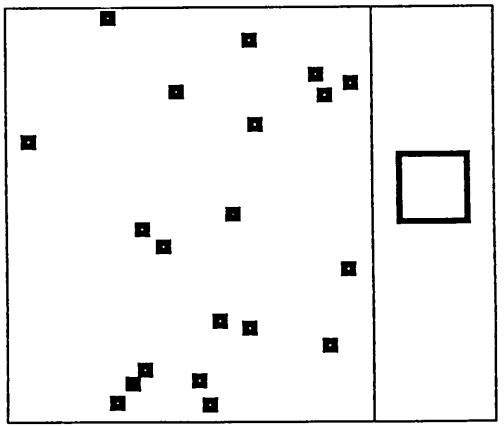
Adding random to the regular

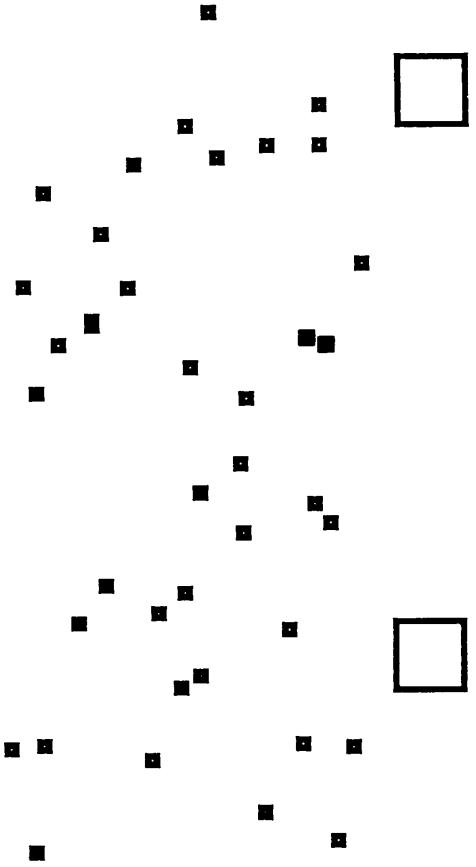
A good place to begin on this venture is to look at the Dutch painter Piet Mondrian. Did you try Exercise 4.10? There I asked you to select a few examples of Mondrian grid painitngs, study them, characterize the grid themes, and translate your characterizations into Logo procedures.

I selected the painting "Compositie met kleurvlakjes nr. 3" of 1917. I am sorry you cannot see the colors—pale blue/gray, oleander pink, and butterscotch—though you can see the shapes and their placement.

Here is what I said to myself. "Those boxes are all upright and almost on a rectangular grid. The dimensions of the boxes themselves are randomly changed in both height and width, but I will simplify that a little, altering them only in the height dimension." How to do it?

Focused random grids #2





"Suppose we start with a regular, rectangular grid. The turtle arrives at a point, but before we draw anything, we randomly 'perturb' the position a little in the x-direction and a little in the y-direction. Then we draw a box that has itself been perturbed a bit in the height direction. Then we go back to the arrival point and let the regular grid machine move us to the next point, and so forth . . . ."

The following procedure mirrors this description. Some samples are shown on the next page, along with the real Mondrian.

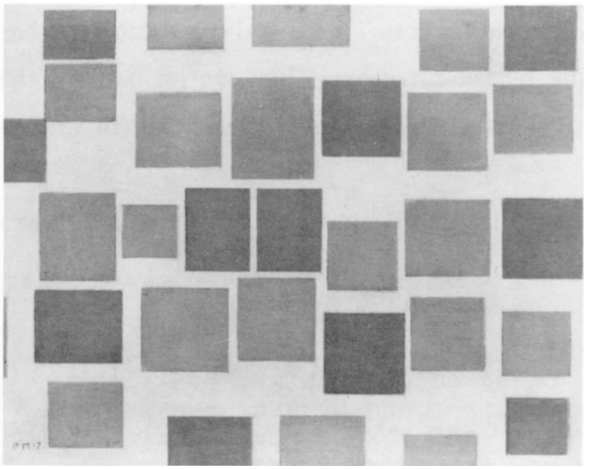
```
TO M.BOX  :E  :DH  :DX  :DY
 ; To draw a Mondrian box based on current x-y position.
 ; The x position for the box is found by adding a random
 ; number from the range -:DX  to +:DX to current x.
 ; The y position for the box is found by adding a random
 ; number from the range -:DY to +:DY to current y.
 ; The height of the box is equal to :E plus a random number
 ; from the range -:DH to +:DH.
 ; The width of the box is always equal to :E.
 LOCAL "HT
 MAKE "HT :E  +  RR  (-:DH ) :DH
 RECORD.POS PU
 SETX (XCOR  +  RR (-:DX) :DX)
 SETY (YCOR  +  RR (-:DY) :DY)
 PD
 REPEAT 2 [FD :HT RT 90 FD :E RT 90]
 RESTORE.POS
END
```

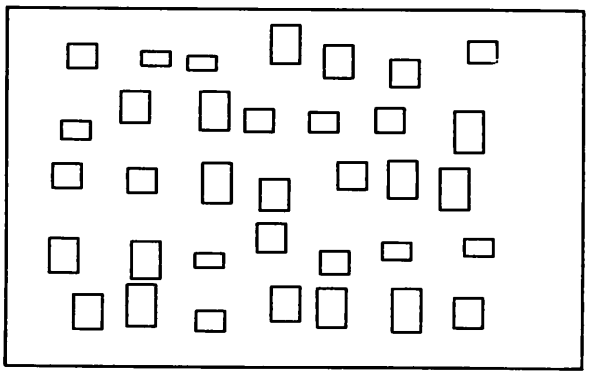
The next operation is to place `M.BOX` —with appropriate arguments—into a `:MOTIF` list and then use `FLAG` to grid the whole affair. Note: `M.BOX` is state transparent.

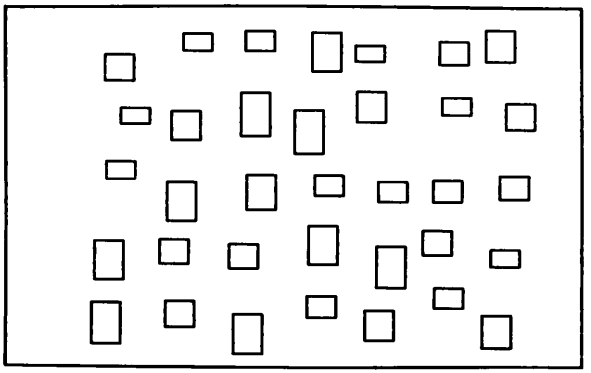
Probabilistic selection of :MOTIFs

The Mondrian boxes exercise was a start toward the integration of random elements into defined geometries. Let's look at one more attempt at this kind of integration before going on to the exercises of this chapter.

A real Mondrian and two grids of Mondrian boxes







Suppose we want to produce a regular rectangular grid from several :MOTIFs, but we want the selection between alternative :MOTIFs at any point to be probabilistic. That is, at any arrival point in the grid, we would like the choice between drawing one :MOTIF or another to be based on given probabilities.

Let's think this through using a model. Define a procedure that selects between two lists according to a probability measure. Let the probability P1 be measured in hundreds units, that is, the number of times an event will occur in one hundred chances. The probability of selecting the first list, :L1, should then be :P1/100, and the probability of selecting the second list, :L2, should be 1 - (:P1/100).

```
TO  P.GET :P1 :L1 :L2
 ; To output one of two lists (either :L1 or :L2) when the
 ; probability of the first being selected is given by :P1.
 ;  :P1 is given as times in 100, e.g., 1 in 100 = .01.
 IF :P1  >  RANDOM 100  [OP :L1]   <--- ???
 OP :L2
END
```

How does this procedure work? The real action comes on the line with the question marks. RANDOM 100 will give one number selected from the range 0 to 99, one hundred possibilities in all. The chance that the number selected will be 0 is 1 in 100 or .01; the chance that the number selected will be 0 *or* 1 is 2 in 100 or .02; and the chance that the number selected will be 0 *or* 1 *or* 2 is 3 in 100 or .03.

Suppose that we set :P1 = 3. Now look back at the marked line above. IF :P1 > RANDOM 100 will be TRUE only if the random number generated is 0 *or* 1 *or* 2. As already stated, this will happen only 3% of the time. Therefore, P.GET will output :L1 3% of the time and :L2 97% of the time when :P1 is set to 3.

Don't be alarmed if you have trouble with these ideas. Probability and statistics are difficult subjects, and our intuition generally doesn't give us much help since it hasn't had much experience in a probability laboratory. Let's give

your intuition a little education on how the probabilistic P.GET operates. Here is a procedure to run P.GET 1000 times to "see" what happens.

```
TO VERIFY :P1 :A
 ; To test out P.GET for any value of :P1.
 ; :P1 is measured in hundreds. So :P1 = 1
 ; would mean a probability of .01 or 1%.
 ; Always give :A a value of 1 to start.
 IF :A > 1000 [STOP]
 IF ((P.GET :P1 [YES] [NO] ) = [YES] ) [PRINT :A]  <--- ???
 VERIFY :P1 (:A + 1)
END
```

Look at the line with the question marks. If P.GET selects the first list, [YES], then :A is printed out. If P.GET selects the second list, [NO], then nothing is printed out. :A is the index that keeps track of the number of times VERIFY has run P.GET. Here is the first experiment. I typed VERIFY 1 1 and the following was printed. Don't be alarmed when you get different numbers for the same experiment. They should be different, shouldn't they?

```
40
114
143
234
300
545
838
850
950
988
```

The interpretation: VERIFY had to run P.GET 40 times before [YES] was selected. It wasn't until the 114th time that [YES] was selected again. The third [YES] occurred on the 143rd time, and so on. In summary, 10 [YES] selections were made in 988 runs of P.GET. That indicates a probability of 10/988 = .01012, very close to 1 in 100. That was just what we wanted to happen.

Let's try it again, this time typing `VERIFY 1 1`. Here's the response:

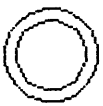
```
18
41
170
277
279
300
412
466
498
707
770
844
863
```

Your turn to analyze these numbers. Change the value of `:P1` and experiment a bit more. Are you getting a feel for how random numbers can help us select things probabilistically?

Probabilistic grids

Suppose we would like to produce a grid that is a combination of the following two `:MOTIF`s:

```
MAKE "L1 [CONGON 30  20  5  2]
RUN :L1
```



```
MAKE "L2 [CONGON 30  20  1  5]
RUN :L2
```

Our task now is to assemble the :MOTIF list. We certainly will use P.GET in the list to select from :L1 and :L2. Let's start with :P1 = 50 (a probability of selecting :L1 of .5 or 50%). You might guess that the :MOTIF list should look like this:

```
MAKE "MOTIF [P.GET  50  :L1  :L2]
```

The reason why this is *not* correct is subtle. Remember when we first discussed :MOTIF lists, we said that the list must contain Logo commands that can be RUN to produce the :MOTIF. The list above does not contain such instructions. How do we get them? We have to run P.GET 50 :L1 :L2 in order for one of the lists that does have the :MOTIF commands to be output. The following *is* correct:

```
MAKE "MOTIF [RUN P.GET  50  :L1  :L2]
```

Here is a little experiment that may help you in understanding the double run business. Type the following lines and analyze the results. How do these results relate to the discussion so far?

```
RUN  [P.GET 50  :L1  :L2]
RUN  [RUN P.GET 50  :L1  :L2]
```

Here is another comparison to explore. Which is correct and why?

```
MAKE "MOTIF [:L1]
MAKE "MOTIF [RUN :L1]
```

Designs

Below are four probabilistic ring grids. The first grid uses this :MOTIF list:

```
MAKE "MOTIF [RUN P.GET 100 :L1 :L2]
```

The second grid uses the same list but with the value of :P1 changed from 100 to 0:

```
MAKE "MOTIF [RUN P.GET 0 :L1 :L2]
```

The third and fourth grids blend the two :MOTIFs according to the list:

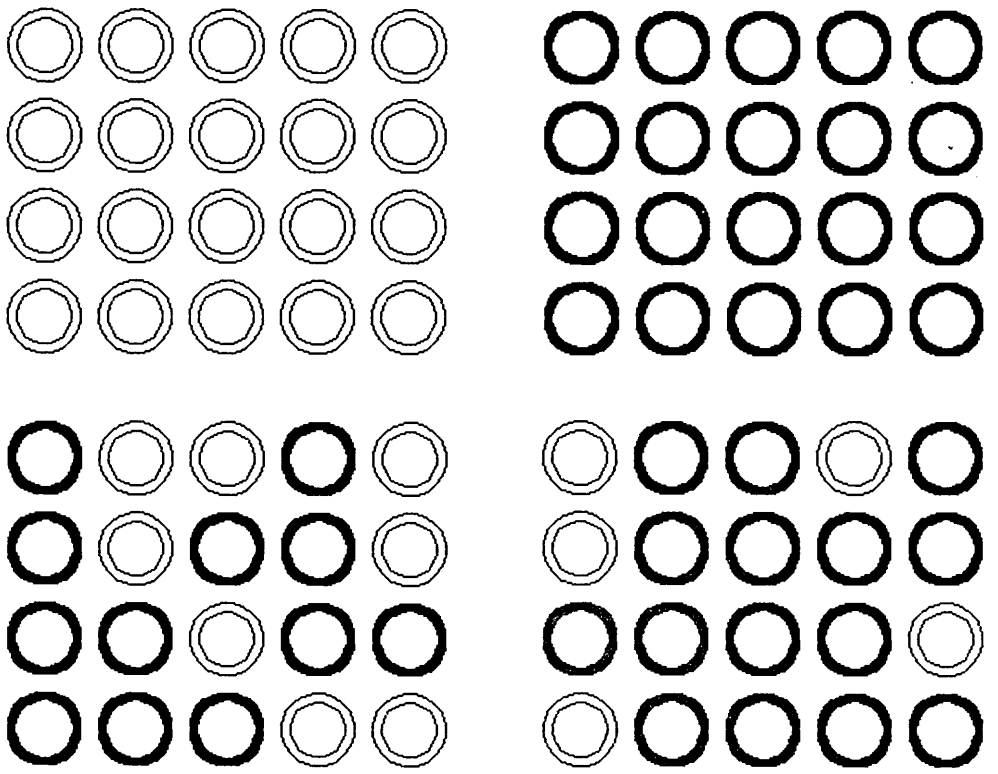
```
MAKE "MOTIF [RUN P.GET 30 :L1 :L2]
```

Think about what ideas are being illustrated here. What does *probabilistic mixing* of :MOTIFs mean in this context? We are introducing some amount of *randomness* into a geometric design. What exactly does randomness mean? On the page after next are some samples that illustrate these ideas using square grids, while those on the page following introduce a new design, a tiny spiral, into probabilistic grids.

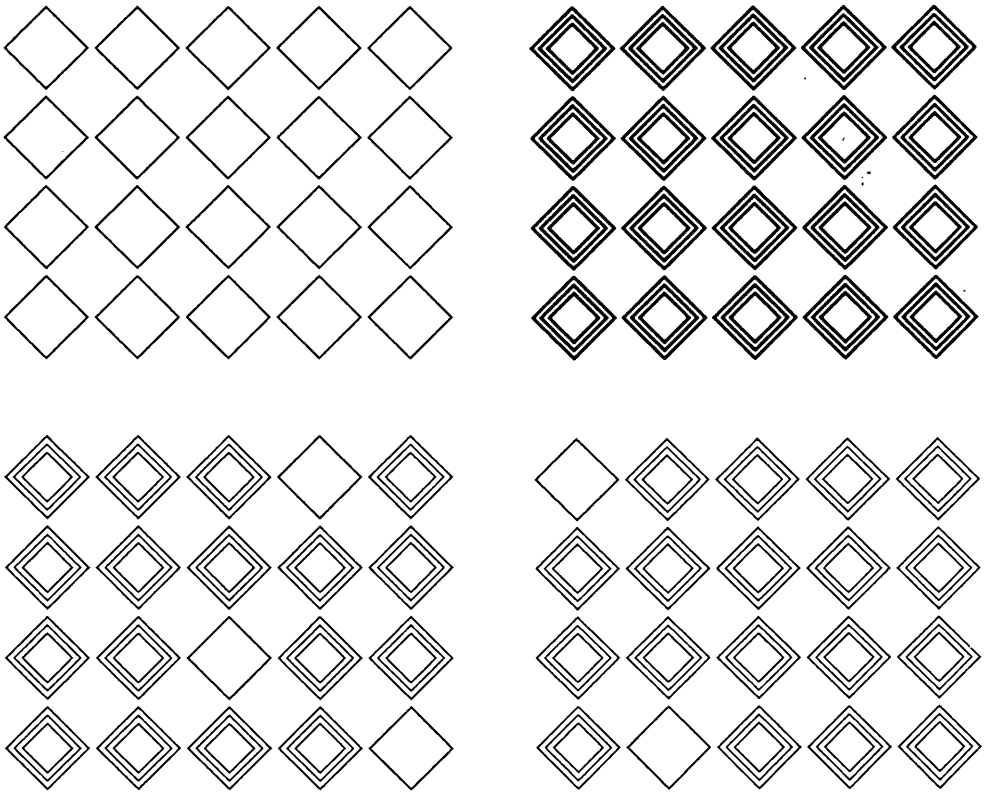
What is important?

This is the longest chapter in the book. We have spent perhaps too much time discussing the machinery needed to build rectangular grids and to generate random numbers. Don't forget, though, that the real goal of this book is to encourage you to use Logo machinery parts to explore the images, patterns, and objects that strike your fancy. Why? Because visual models can extend and amplify your vision, and I am certain that this will give you an enormous amount of aesthetic satisfaction. Be careful, though, not to get too involved with *only* the model parts; the model, after all, is always to be *aimed* at the object of your visual fancy.

Probabilistic ring grids

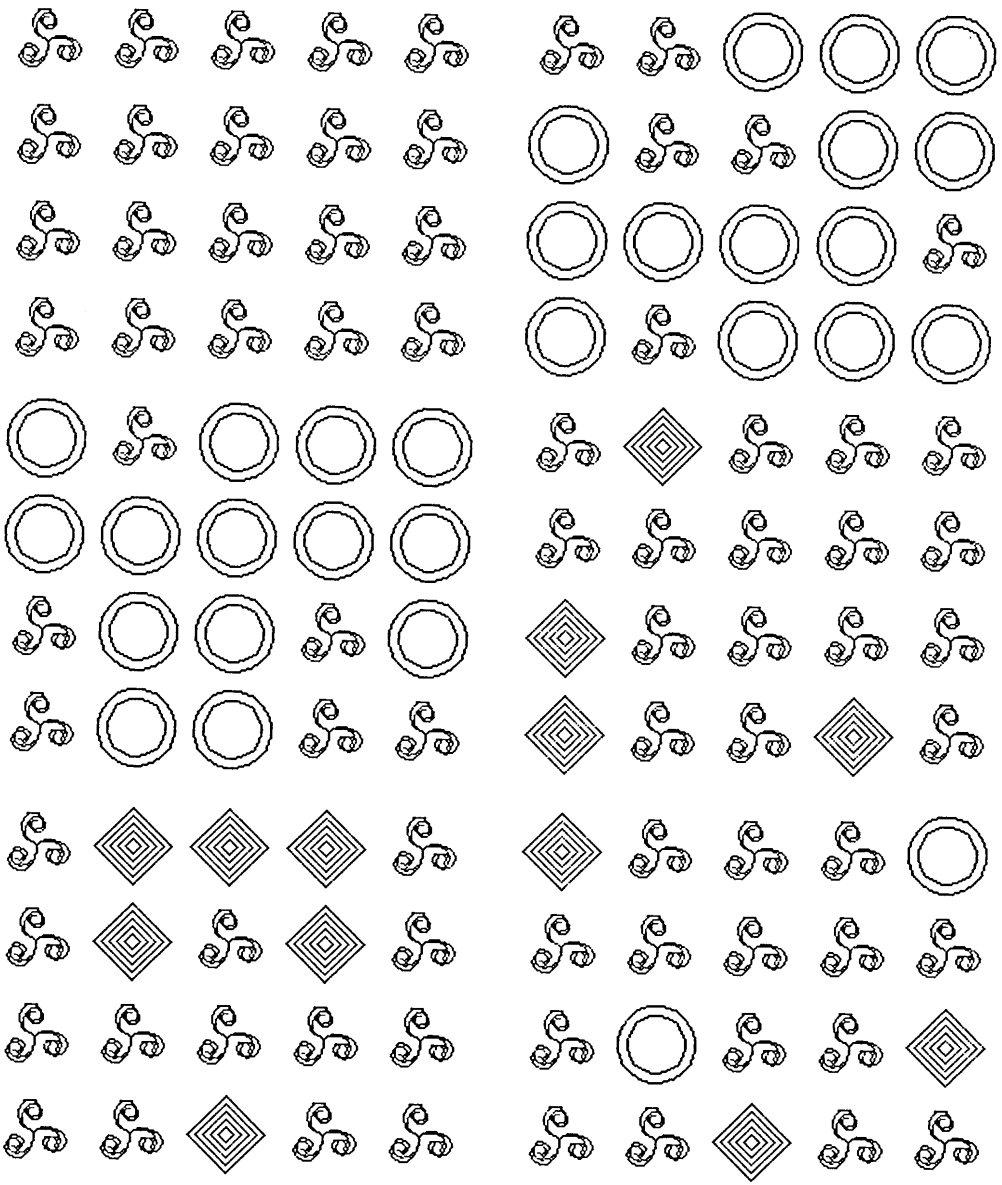


Probabilistic square grids



Probabilistic grids with spirals, rings, and squares

The little spirals should be easy to reproduce. Can you guess at the `:MOTIF` lists used for all the designs? What is happening in the last `:MOTIF`? There are three `:MOTIF`s. How could this be structured? Exercise 5.5 will ask you to attack this problem formally.



Exercises

There are eighteen exercises this time. They are not so easy, but you will be surprised, I hope, by how quickly ways to start will pop into your head. Don't forget the turtle-walk/story-board approaches to developing intuitive brain-pops, though.

Exercise 5.1

Go back and look at the circular grids you produced in Chapter 4. Review, as well, the personal mark that you created for Exercise 3.6. Add some randomness to these designs.

Exercise 5.2

Restructure the FLAG procedure using recursion techniques. Avoid any use of the REPEAT command. Compare the two methods on aesthetic grounds. You might want to define carefully in words exactly what you mean by "aesthetic."

Exercise 5.3

The rectangular grid procedures that we have studied so far place one row directly below another. Design a more general FLAG procedure that allows the even-numbered rows to be indented by some given amount. Use either repeat or recursive methods.

Exercise 5.4

Produce a super, razzle-dazzle demonstration of designs never seen before by Earthlings using the ideas of this chapter: motif descriptions with lists, placement of these motifs using variations of geometrical repetition rules, random number generation, and probabilistic list selection.

During the preparation of this exercise, jot down any *design rules* that you discover. Illustrate your rules with examples of designs that "work well" and ones that "don't work at all."

Exercise 5.5

Generalize `P.GET` so that it can select any number of lists. Extend the definition of :P1 so that it can specify the probability of selecting *each* of the possible lists. Use your generalized `P.GET` to produce a number of multiple-`:MOTIF` designs.

Exercise 5.6

Although we have designed a procedure called `FLAG`, we haven't designed any flags yet. Redesign your own country's flag so that it better represents current life there—as you see it. You might want to find a book on heraldry so that you can break all the established rules and traditions. Or not. Either way, keep the iconography simple. Describe your heraldic symbols and symbolism in your notebook.

Exercise 5.7

If you have a flair for flag modeling, design a generalized, multiple-argument, flag-machine that will design a flag for *any* country once the appropriate argument values are established for the country.

Exercise 5.8

I found this exercise pinned to the wall of a classroom used for a Color and Design course: "Channel 8 is preparing for an eventual slot on French television. One of the first pieces of artwork they need is a TV test pattern. The main function of a test pattern is to show technicians the quality of picture resolution and the fidelity of the colors being broadcast.

"Test patterns usually contain some kind of black and white value scale composed of varied linear patterns of thick and thin lines, as well as letters or numbers to provide detail in judging the broadcast image. Of course, another role of a test pattern is to tell the viewer that he has tuned into Channel 8 even though they're not broadcasting at the time. So, include three character call numbers—CH8—in the design, but not as the major design element.

"Traditionally, test patterns have been designed around a central circular format, but there is no need to follow this tradition. Be sure to fill the rectangular screen, too."

What can you do with this? Take advantage of the strengths of Logo and design a test pattern that could not have been realized in any other medium.

Exercise 5.9

The illustration on the next page is Andy Warhol's "Marilyn Monroe," done in 1962 during the Pop Art movement. You may recall Warhol's rectangular grid of tomato soup cans. Each of the images of Marilyn, and each soup can, is exactly the same; this sameness was the message of Warhol's design.

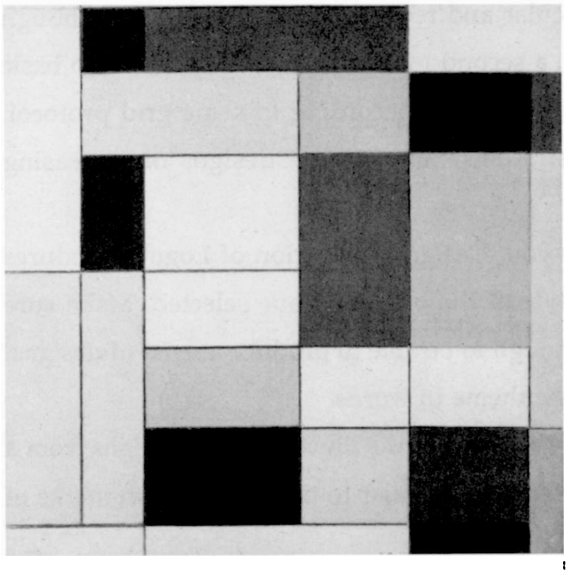
I want you, however, to produce a grid of stylized faces, whose characteristics (openness of eyes, openness of mouths, tiltiness of eyebrows, etc.) are randomly *perturbed* from one face to the next. (Recall the Mondrian box exercise.) Your faces will exhibit a series of different expressions. Don't use too many characteristics—maybe two or three—before you visually experiment.

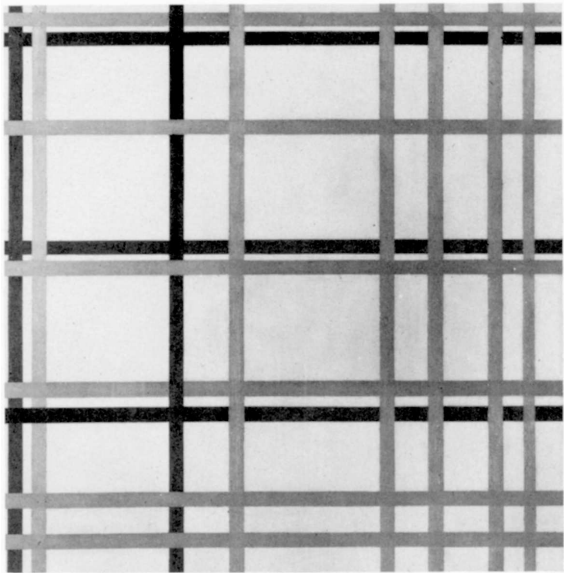
Exercise 5.10

I love Mondrian. We simulated several of the characteristics of one of his paintings in this chapter. I want you to do more analysis of this sort. I include several examples of Mondrian's use of vertical and horizontal bars to articulate a canvas. Some of the rectangles produced by these bars are colored in, and others are not. I want you to select several characteristics from these paintings and simulate them with Logo procedures. Clearly state, in words, the characteristics you are studying.

This kind of exercise in no way trivializes the work of a Mondrian. Rather, you should have a better sense of his artistic genius after you have tried to copy paintings like these.

Two more Mondrians





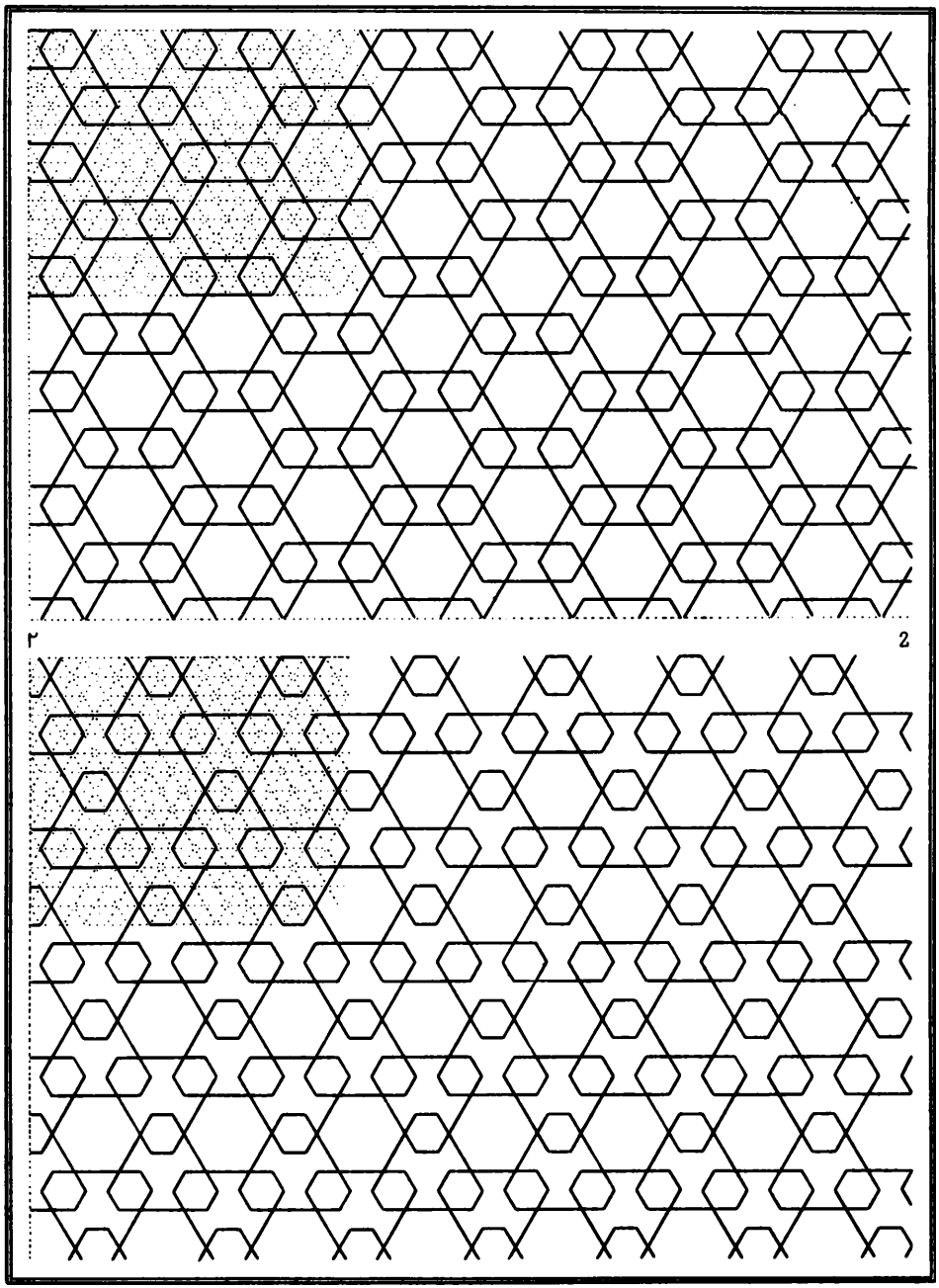
Exercise 5.11

Islamic art offers a treasury of circular and rectangular grid designs. Although at first glance many look complex, a second look will uncover one or two basic motifs that are repeated over and over again according to some grid protocol. Look through the following illustrations; they exhibit designs of increasing complexity.

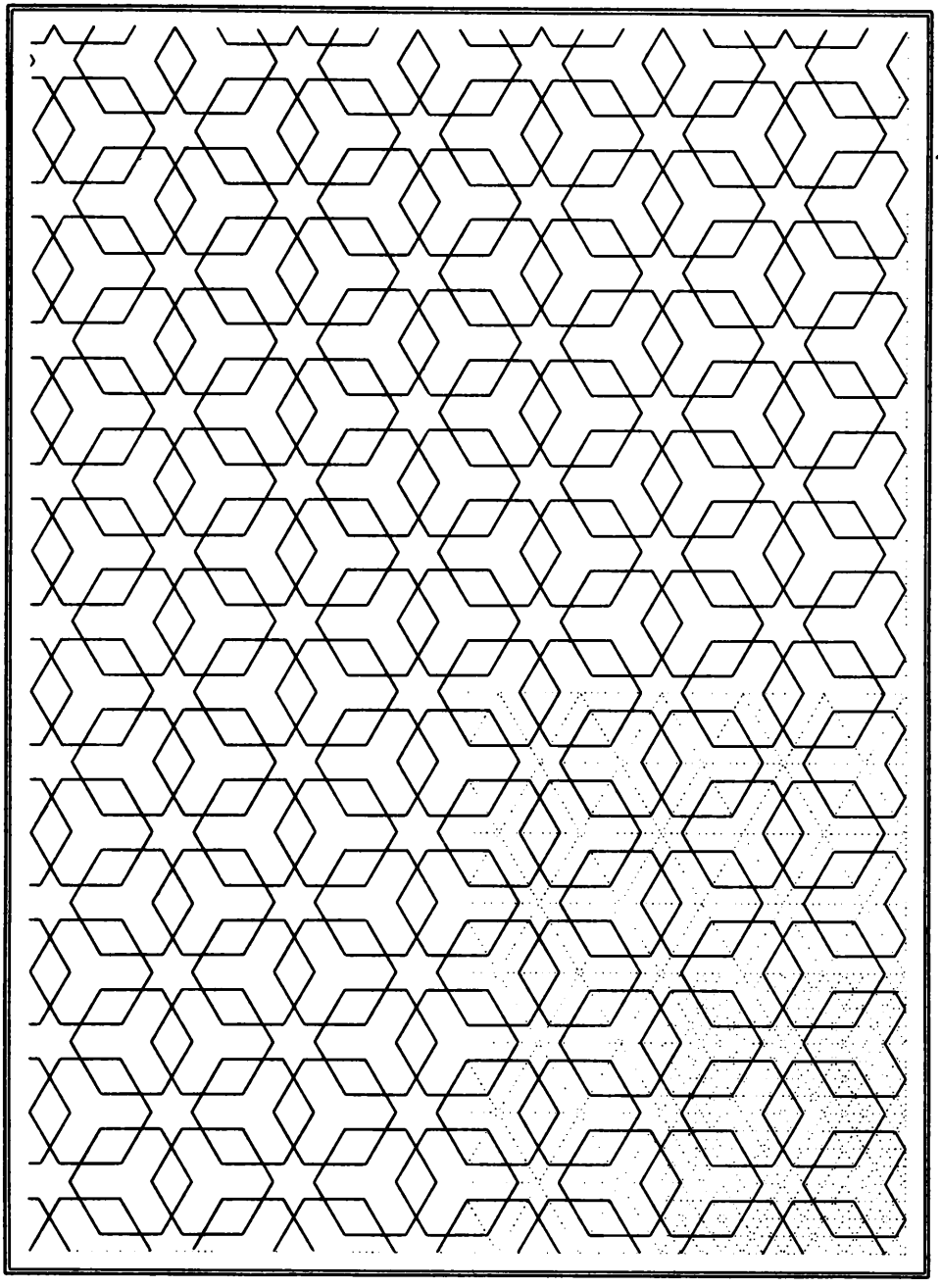
Find a design that appeals to you. Design a collection of Logo procedures that can produce designs in the style of the one you have selected. Make sure that your procedures are general enough to be able to produce a *series* of designs, all based on a single theme. State the theme in words.

You might be able to produce a number of the illustrated tile designs from a single Logo procedure. This exercise is very similar to the stone mason marks of Chapter 4.

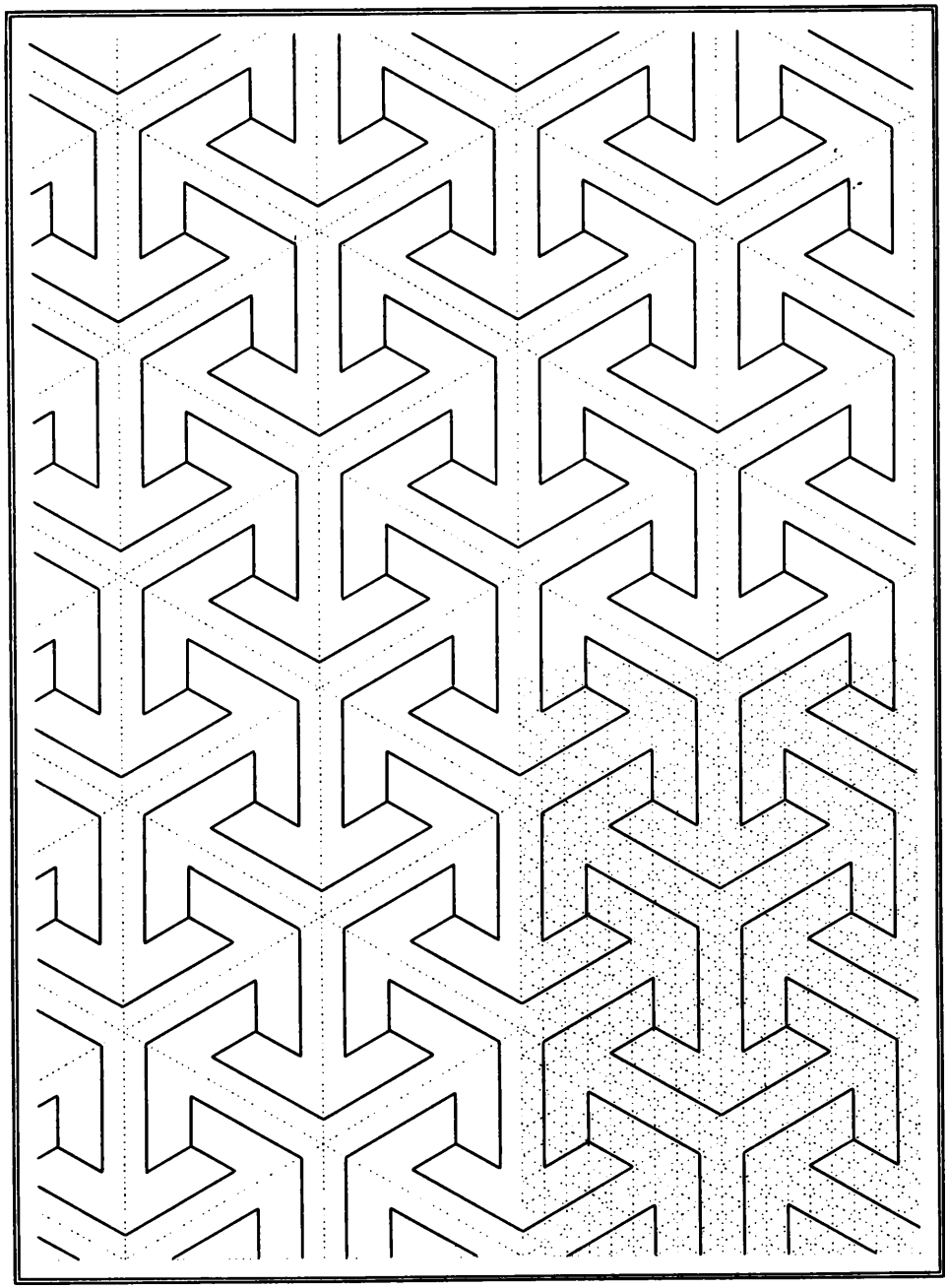
Islamic tile design #1



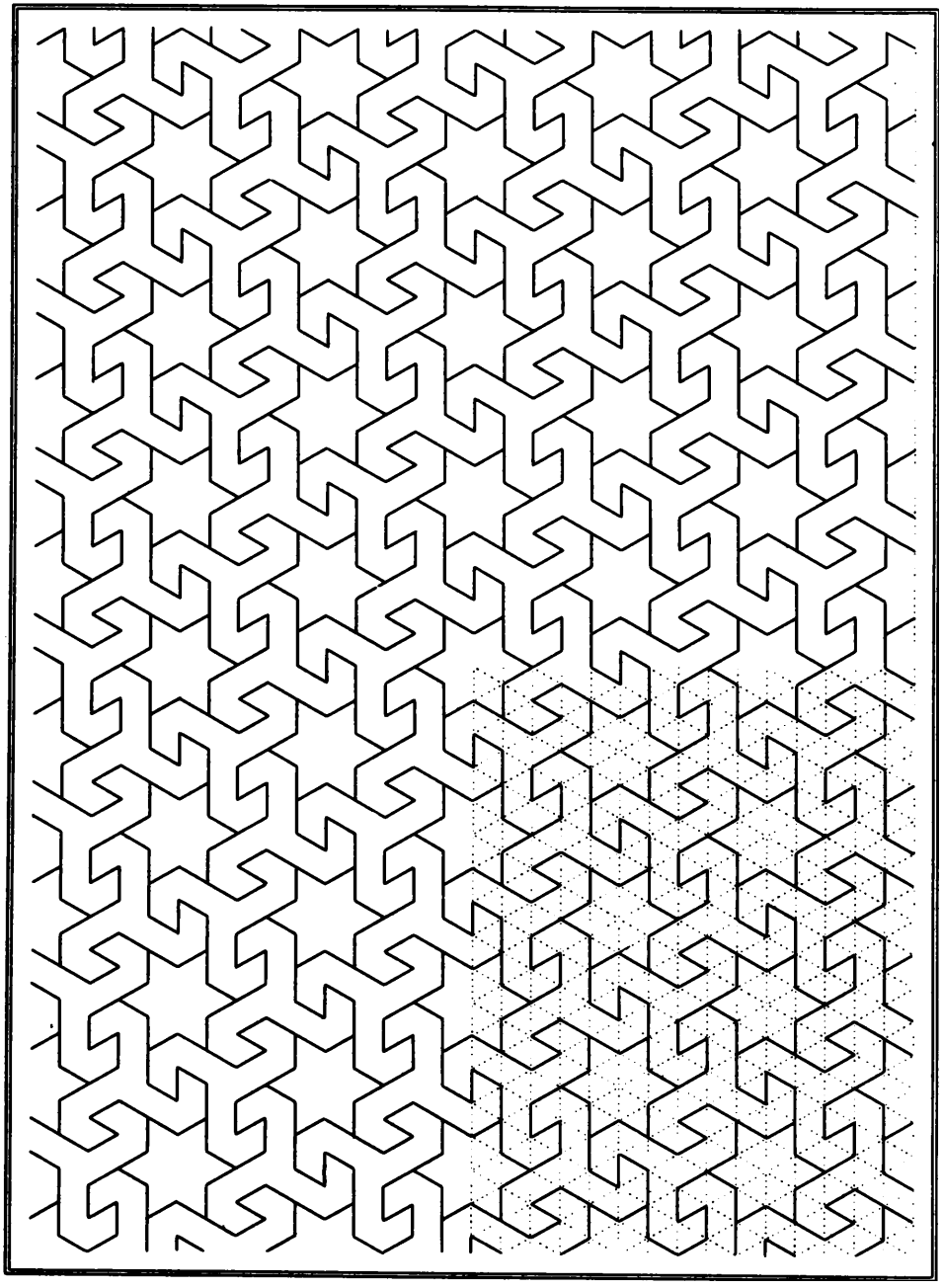
Islamic tile design #2



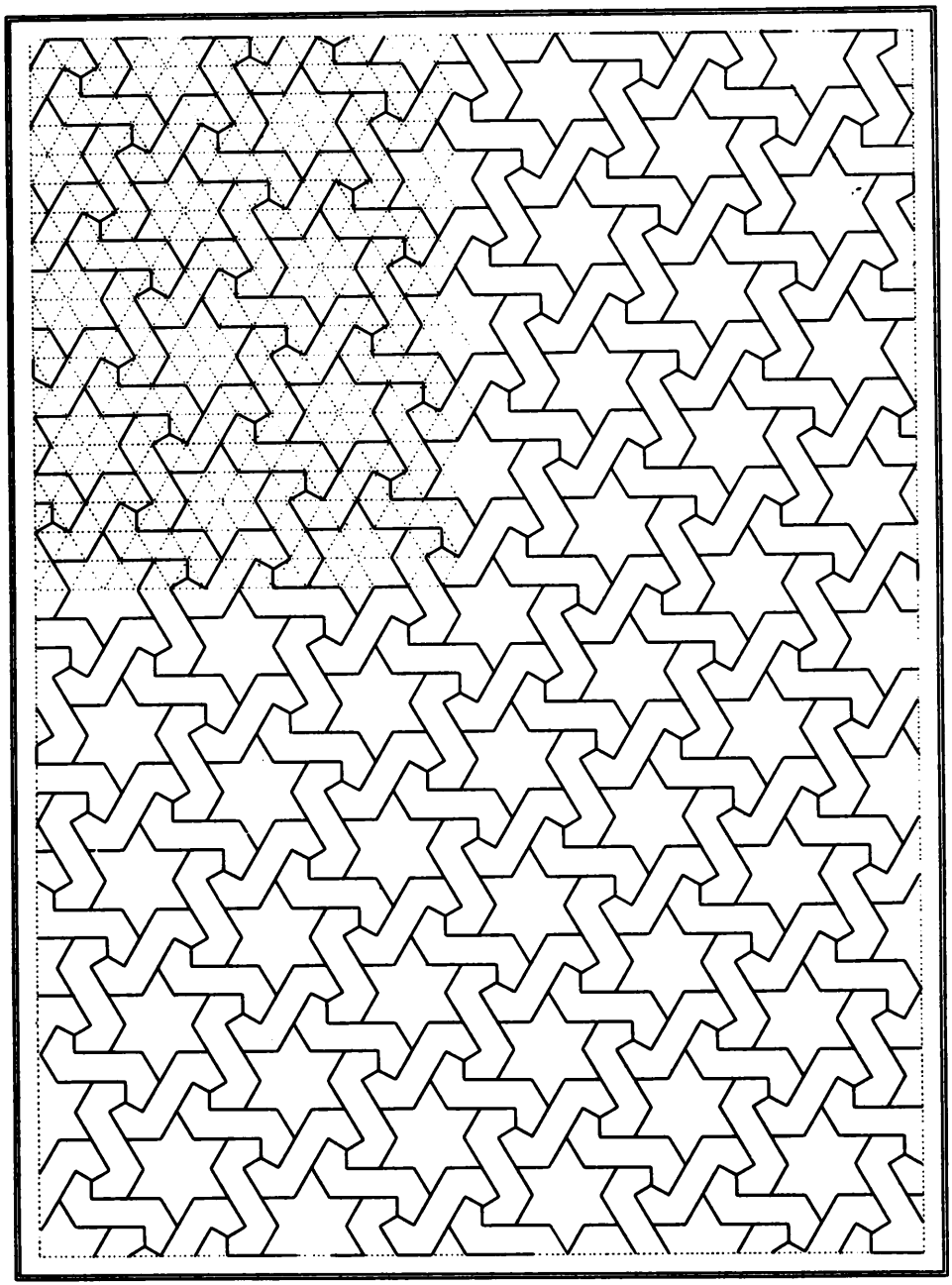
Islamic tile design #3



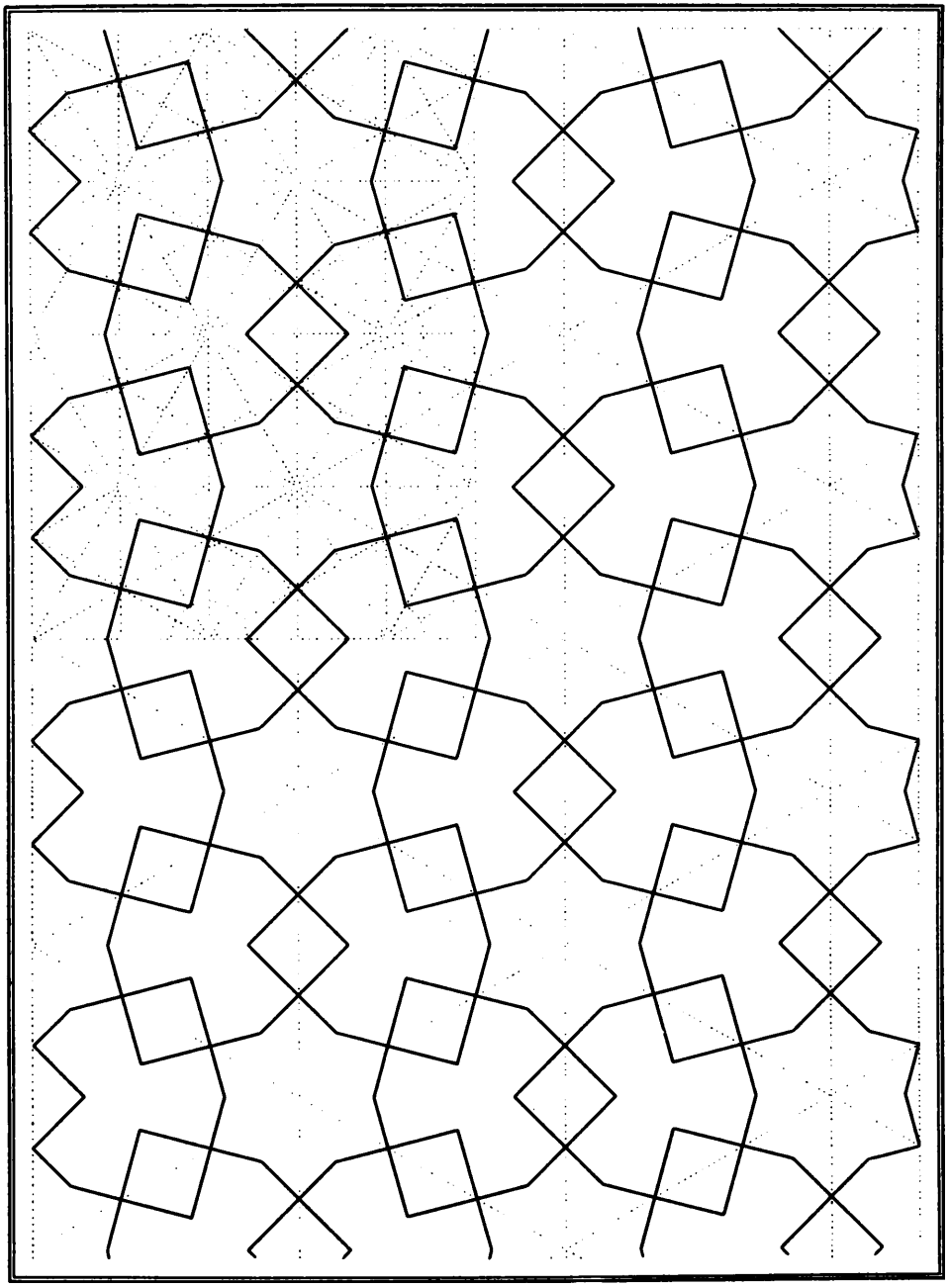
Islamic tile design #4



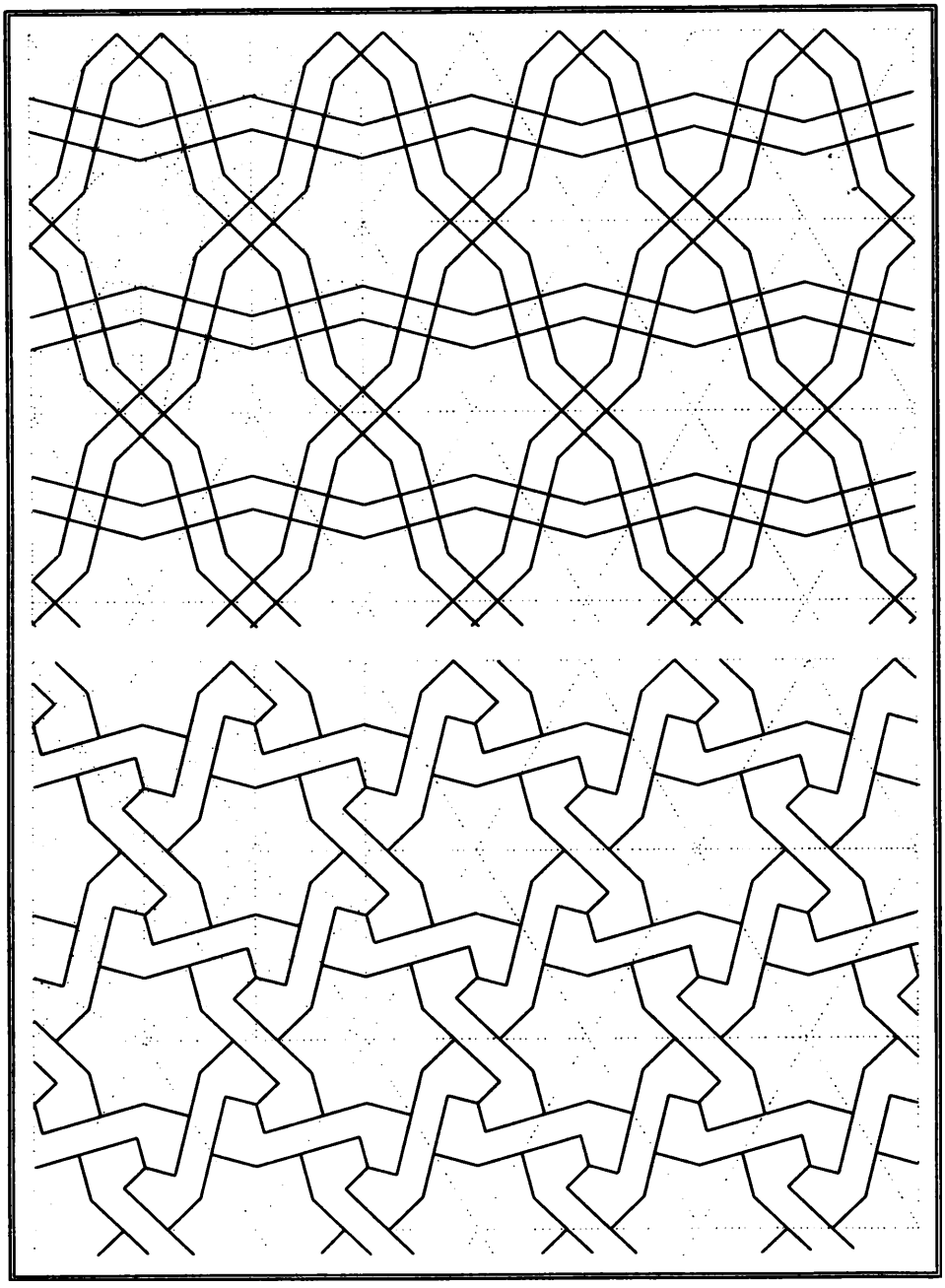
Islamic tile design #5



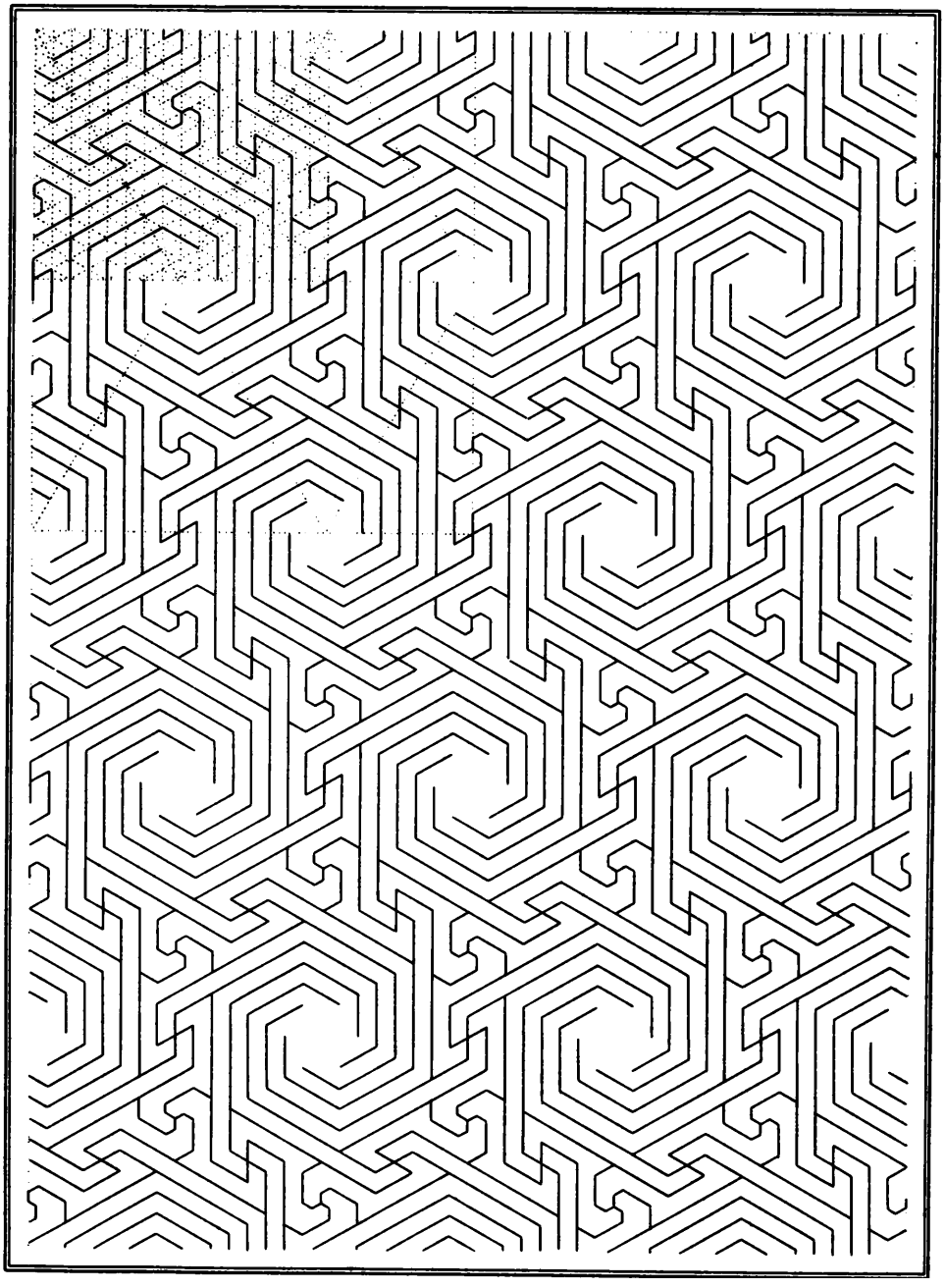
Islamic tile design #6



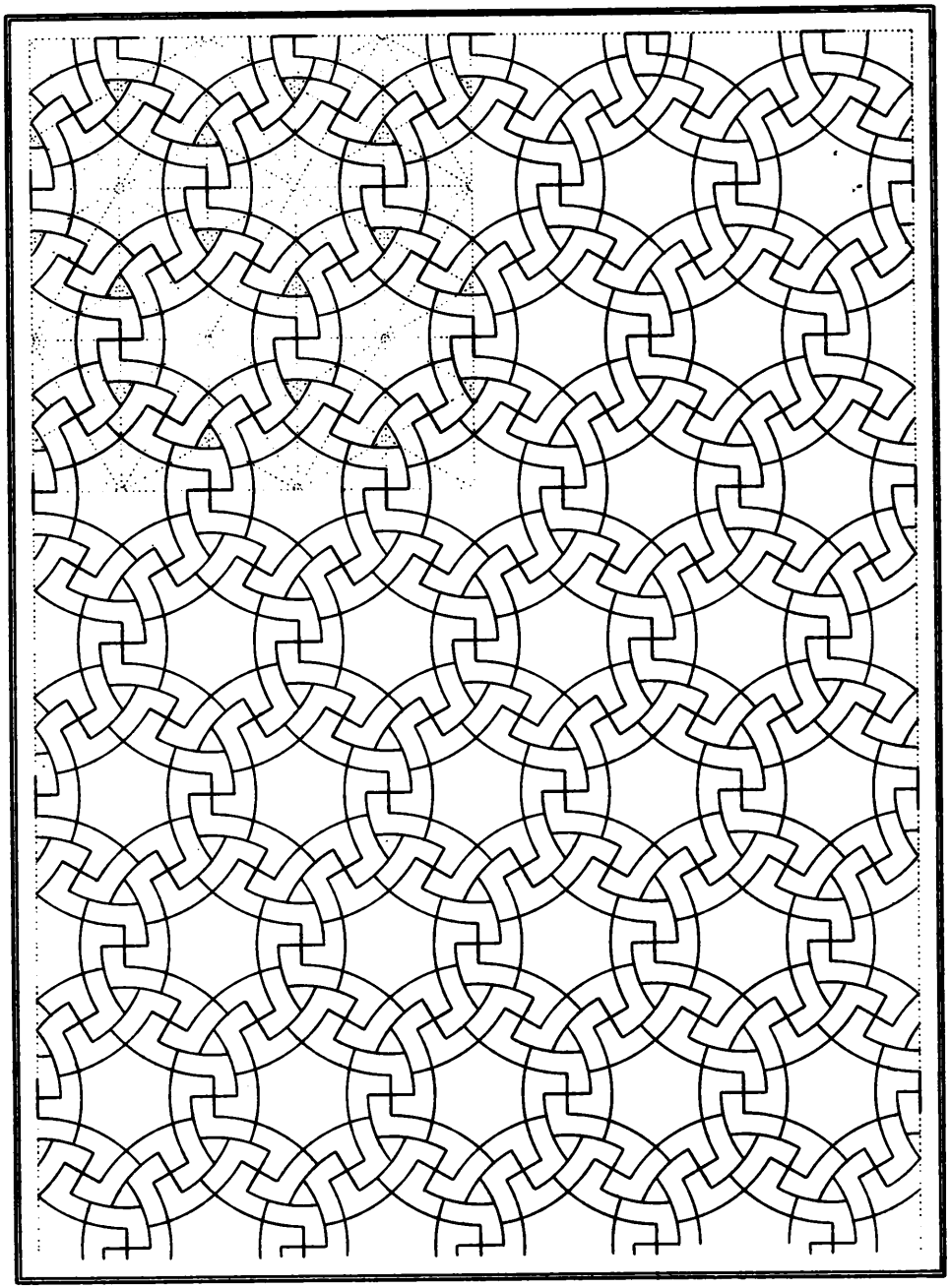
Islamic tile design #7



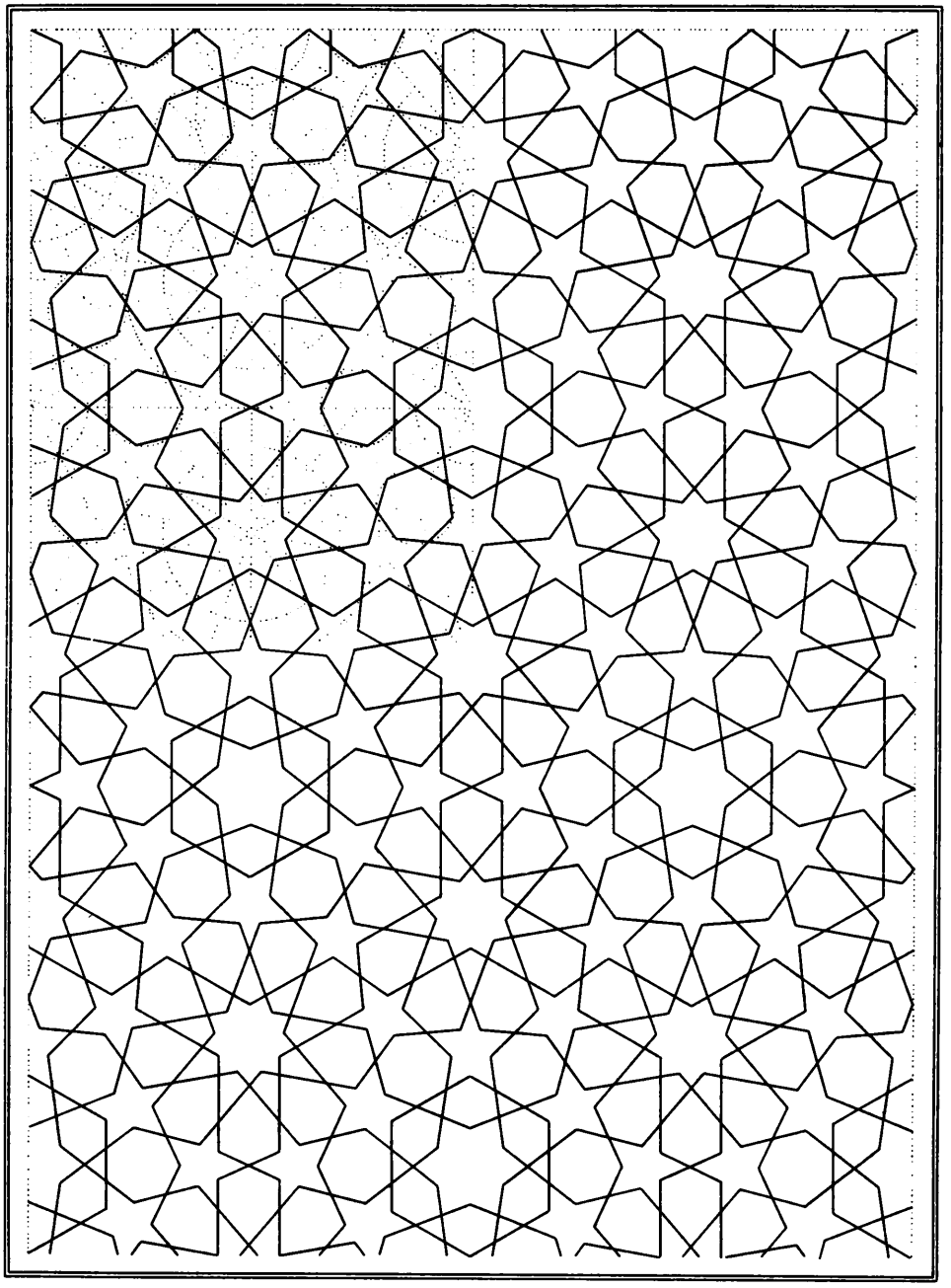
Islamic tile design #8



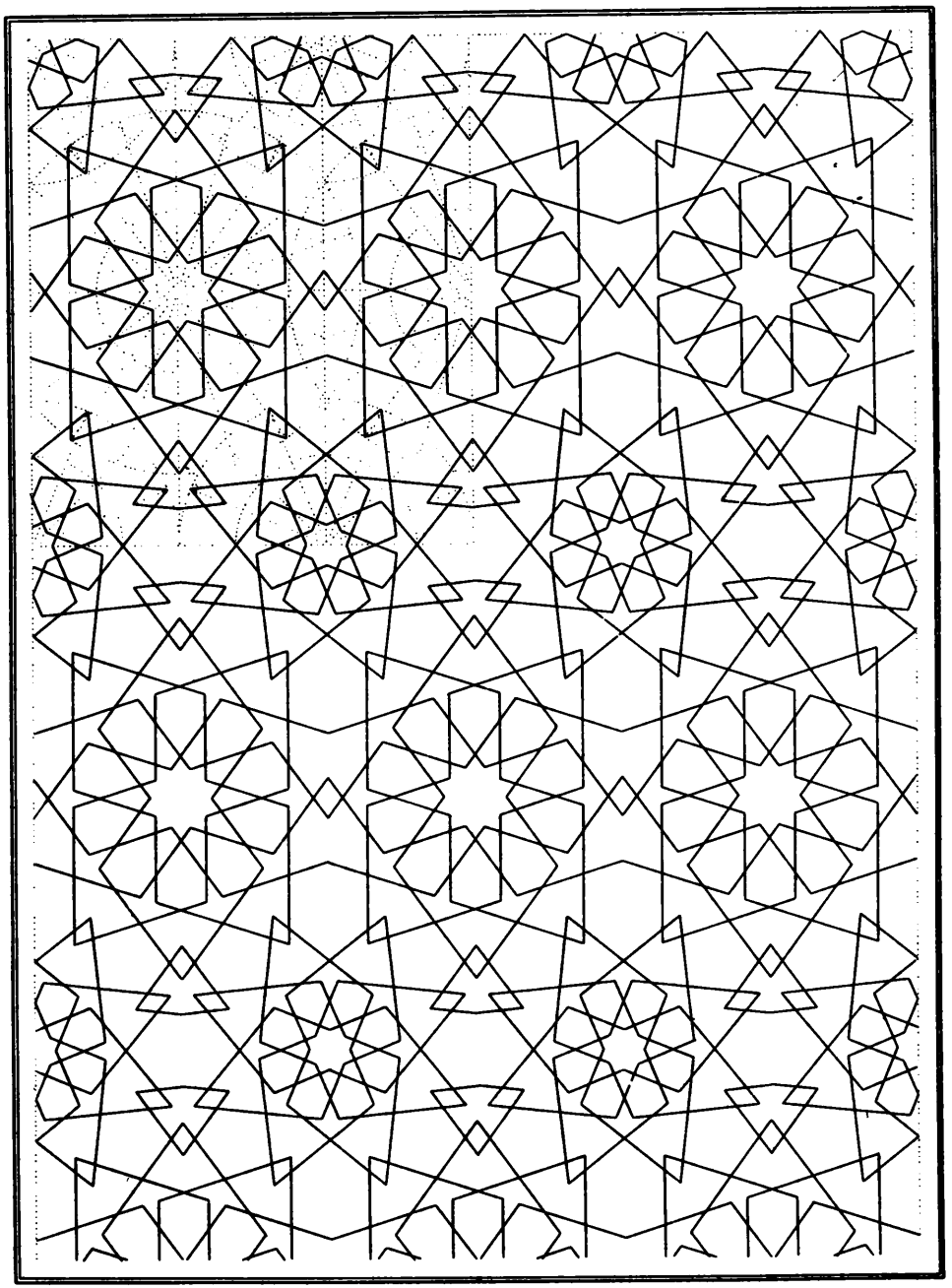
Islamic tile design #9



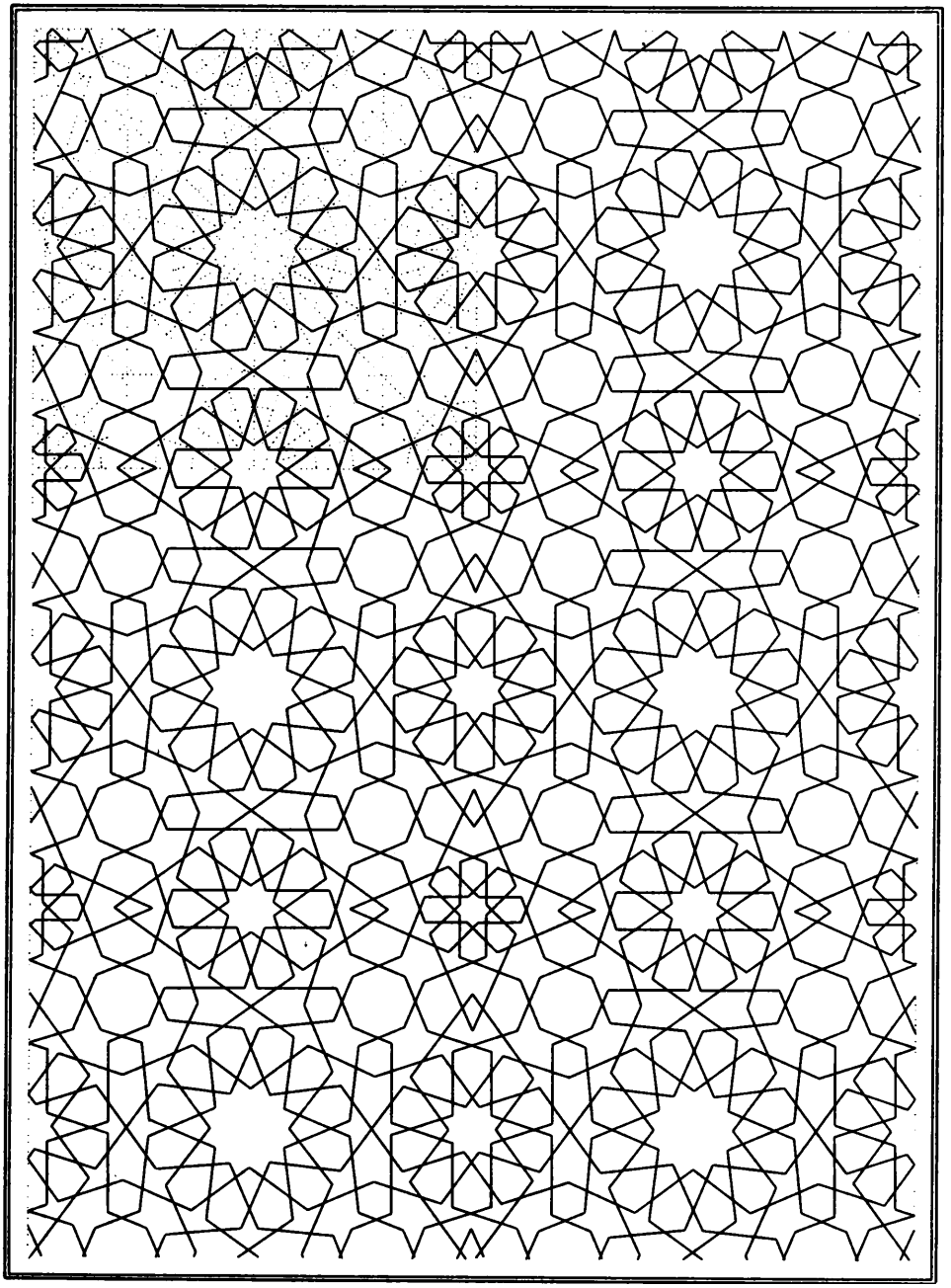
Islamic tile design #10



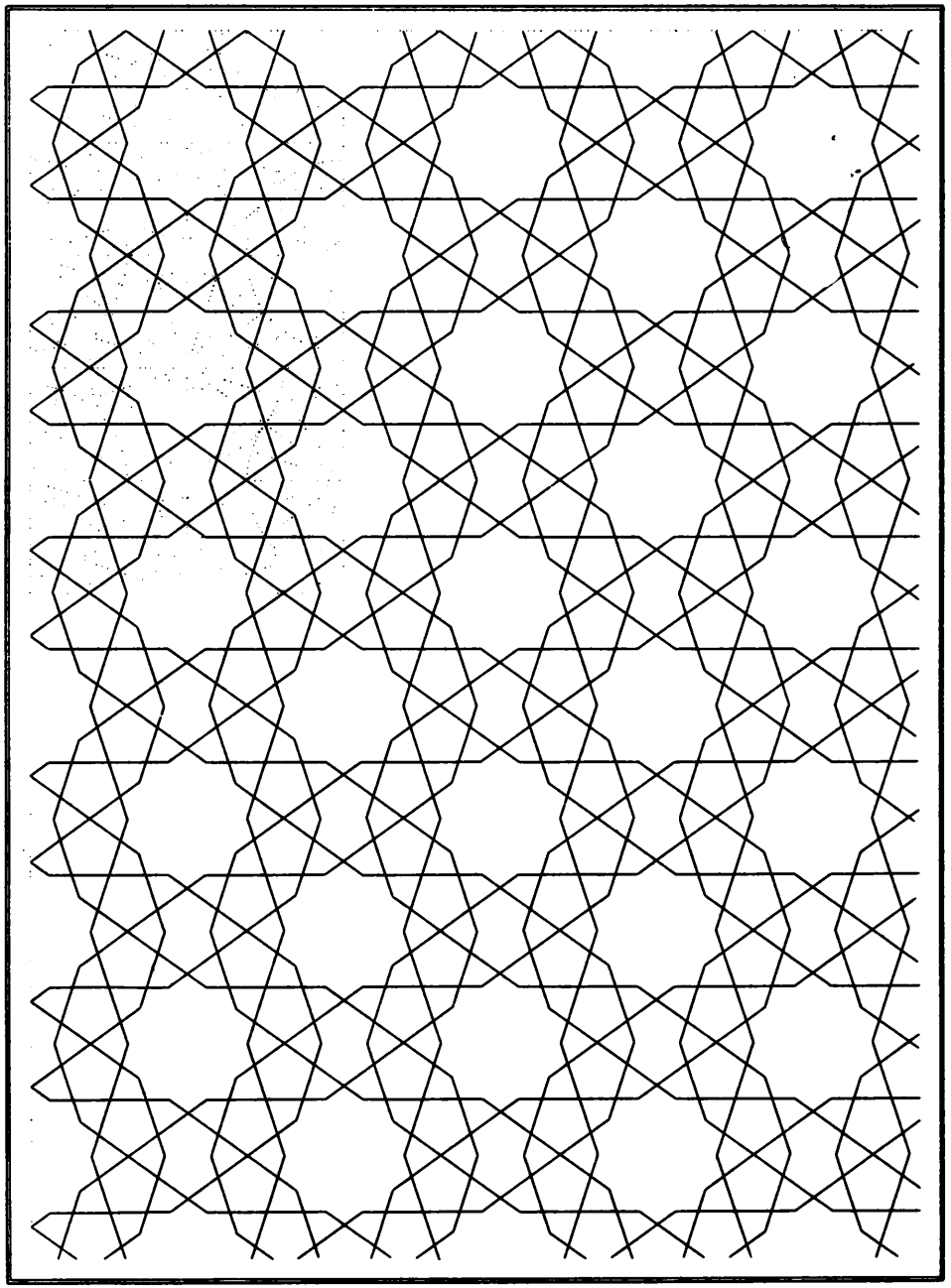
Islamic tile design #11



Islamic tile design #12



Islamic tile design #13



Exercise 5.12

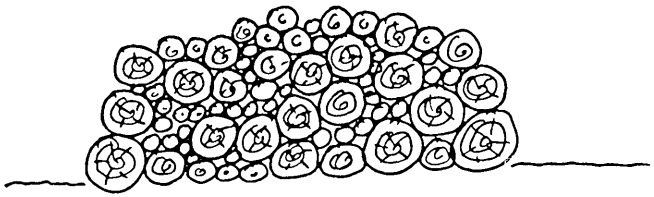
Are you beginning to see grid patterns everywhere?

This chapter has suggested that grid-thinking is a convenient method for finding and analyzing regular patterns. Furthermore, our work with random components suggested that grid-thinking can be extended into the realms of irregular shapes and forms. The design materials used in the chapter, however, were selected from the *created* worlds of the artist and designer. The final two exercises of this chapter suggest that grid-thinking can also be useful in describing ways of seeing irregular patterns in the natural world.

I have a small, seventeenth-century house located in the department of the Sarthe, about 100 miles southwest of Paris. The Sarthe, nestling between Normandy and the Loire Valley, is covered with forests, fields of apple trees, ponds, streams, and marshes. During the long and mild Sarthe summer and autumn, the meadows surrounding my tiny house are filled with the ochre and black and white cattle that are typical of this region of France.

But most typical of the area is the ubiquitous wood stack. Every Sarthe farm has at least one, and each stack grows larger after the annual harvest of farm trees and hedgerows. And even though many farms are no longer heated by wood, the Sarthe farmers continue to build new stacks. Although I have often noticed these wood piles in a casual way, it was only recently that I began to sketch them and to think of them as sculpture. Suddenly, I realized that these stacks are magnificent "wood works"; in fact, they are huge *log grids.*

On the next page is a quick sketch of a stack of logs. What rules might have been used to decide how to stack the various sized logs into this big pile? Design some Logo procedures to generate log grids based on *different* stacking rules.



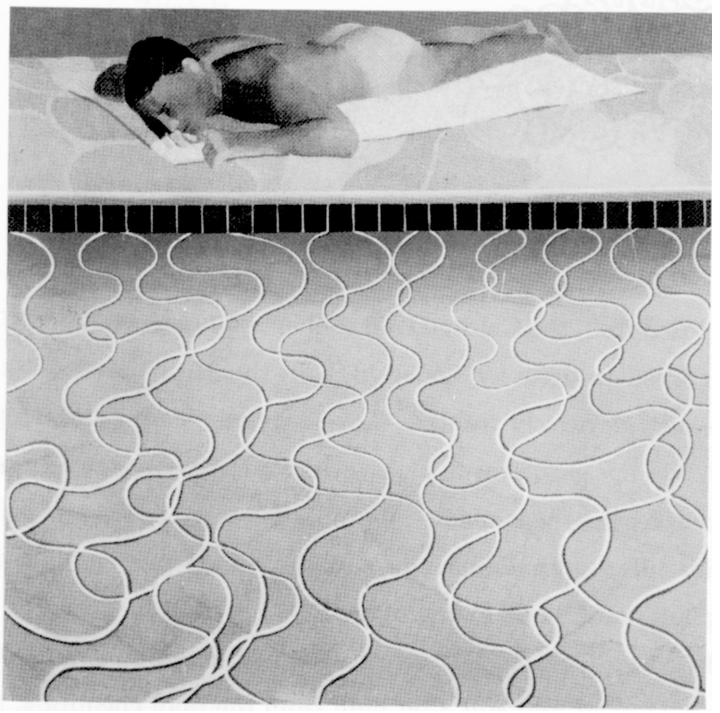
Exercise 5.13

Twenty-five years ago I fled from the Chicago cold to the continuous summer of Southern California. I lived in a funky house high in the Hollywood Hills overlooking polluted—yet gloriously lighted—Los Angeles. Alas, I suffered from one great loss. While my garden was lush with tropical plants, I had no pool. My sense of incompleteness was made only more bitter by the fact that I looked down on a city filled with thousands and thousands of pools. The impression of distant pools is still vivid in my mind.

Last summer I visited my Pasadena mother-in-law *and* her pool. One lazy afternoon, sitting in a deck chair and still a bit high from the pool chemicals, I studied the patterns created inside that pool by LA light hitting the rippling crystalline water. Yes, it was pool-light grids that I saw; and one Pasadena pool was suddenly different from all the other pools.

Unfortunately, you may not have a pool at your disposal to study, even one at a distance. As an alternative, the basins of fountains are good; they contain clear water with plenty of surface disturbances. If you lack both pools and fountains, I suggest that you look at a few paintings by David Hockney. I know of no other painter who has so well caught Southern California pool light.

Look closely at the following Hockney painting. Does it suggest an approach to designing watery grids? Can you generate a series of pool-light grids that are all of your own making? But maybe you would rather describe the reflective patterns seen in rivers, streams, or lakes?



Exercise 5.14

If you attempted the Hockney grid exercise, you may have found yourself needing an arc drawing procedure. We can draw circles, but what about arcs? Design a left and right drawing arc machine that takes two arguments: the radius of the circle of which the arc is part, and the angle of the arc. Hint: the circumference of any circle is equal to $2\pi R$, where R is the radius of the circle. The length of an arc is equal to the circumference of the circle from which it is taken times the angle turned by the arc divided by 360 degrees.

Exercise 5.15

Every season of the year has some major festival. Use Logo to help you design an appropriate greeting card for a fête that is coming soon. Don't forget the envelope; your card and envelope designs must complement each other.

The only rule for this exercise is this: produce a card that could not have been easily or appropriately realized in any other medium. You can combine Logo designs with other materials, however, and collages are fine. You might want to add visual remarks with paint, inks, photographs, glitter, whatever, to your Logo-produced images. Or you may want to cut away parts with an Exacto knife. You can photocopy your prints onto different colors and strengths of paper, and you can build cards into the third dimension. But whatever you create, it must reflect the style and flavor of Logobuilt visual models.

Exercise 5.16

We have worked only with rectangular windows in this chapter. Suppose you wanted to place a motif randomly into a triangular or circular window. How could you do it?

Exercise 5.17

Suppose you wanted to place a motif randomly into a rectangular window, but suppose, too, that you did not want the motifs to overlap within the window. Any ideas?

Exercise 5.18

Are you exhausted by all this? Or excited, uplifted, depressed, furious? I think it's time for you to do some visual modeling of your moods and subjective feelings. Pick an emotion from the list that follows and design a Logo machine to produce images that correspond to your idea of this particular state of being: *Love, lust, greed, enthusiasm, envy, jealousy, happiness, joy, aggressiveness, lethargy, alertness, hunger, guilt, fury, anger, compassion, generosity, arrogance, passivity.*

In preparation, you may want to jot down the ideas, shapes, colors, textures, and images that come to mind as you freely associate this emotion with the contents of your mind's baggage. As you think about your selected emotion, try to imagine how you feel about its opposite. Thinking about an opposite or reverse state can often focus your vision of the original state.

Let yourself go with this exercise, but be concrete; capture your ideas and feelings graphically. Sketch fast; grab the images as they float by. Don't worry about how you will explain to others what you have done. Describe simply what you see inside yourself with Logo procedures.

Consider the following. Include a two-way switch on your emotion modeling machine. One switch position should direct your machine to illustrate the positive side of an emotion, while the other position will show the negative or reverse side of it. Instead of a switch, you might be happier with a dial that would let you illustrate a series of emotional states between, say, love at one extreme and hate at the other.

Chapter 6
Islamic Designs

"Know, oh brother . . . that the study of sensible geometry leads to skill in the practical arts, while the study of intelligible geometry leads to skill in the intellectual arts . . . . "
From the "Rasa'il," translated by S. H. Nasr.

Islamic designs combine circular and rectangular grids

In the last two chapters, we have been working on circular and rectangular grids. Within these chapters, the major exercise was to decompose given designs into their basic components so that these parts could be modeled as generally as possible. We then went on to build models to place these parts into forms that were either circular or rectangular. We referred to these two design aspects as the motif and the repetition rule.

The goal of this work was to build models that produced not just a single design but a *suite of designs* all based on one or more themes. The exercises at the ends of these chapters encouraged you to take your design apparatus outdoors, into distant fields, to find and sketch more exotic subjects. Now we turn to Islamic art, an exceptionally rich source of sophisticated geometric designs. We will use a selection of these fascinating patterns to test your modeling style and then to extend it into new dimensions. Indeed, we will end the chapter with three-dimensional designs.

Much of the complexity sensed in Islamic designs is created by a combination of circular and rectangular repetitions of several design motifs. In fact, these combinations are often multilayered and recursive: circular grids are ordered into rectangular patterns that are in turn combined into larger rectangular designs. Our job is to make this geometric complexity intelligible, and modeling is our most useful tool.

The ability to place individual images and clusters of images into rectangular patterns is crucial to effective Islamic design work with Logo. We need, therefore, a really general and easily manipulated grid machine. Toward this end let's begin this chapter by generalizing the FLAG procedure of Chapter 5. We will be using this procedure for the rest of this book, so we had best make it as useful as possible.

Exercise 5.2 asked you to restructure the FLAG procedure using recursive rather than REPEAT methods. Exercise 5.3 asked you to design a more general FLAG procedure that would allow each row to be indented in relation to the starting x position of the first row. Let's combine these two exercises into one. Additionally, let's allow *each* row of the FLAG to have a different number of columns and a different indent amount. But first we need a few more list manipulators. They will be useful in the refashioned FLAG.

Rotating a list

We need another list manipulator to refashion FLAG, and here it is. Suppose you have a list of items. How might you form a new list whose last element was the first element of the original list, whose first element was the second element of the original list, whose second element was the third element of the original list, and so on? You could think of this change as a kind of rotation. *Rotate* the first element of the original list to become the list's last element; keep all the remaining elements as they are, and then output the list. Here is a procedure to accomplish this list rotation:

```
TO ROT :LIST
 ; To put first element of a list at the end of the list.
 OP LPUT FIRST :LIST BF :LIST
END
```

Note the presence of the command OP. What happens when the argument given to ROT is a single-element list? When the argument is the empty list?

A generalized rectangular grid machine

I think you will understand the following procedures as you read them. I believe, too, that you will find the recursive form of FLAG to be easier to read than the REPEAT form introduced in the last chapter. Do you think that "easier to read" is the same as more aesthetic?

Here is a summary of the arguments of the revised procedure called RIFLAG in which the RI stands for recursive and indented:

:COLS denotes column information and must be a list. Each element of :COLS indicates the number of columns per row. For example, [3 2 1] indicates that the first row has 3 columns, the second row has 2 columns, and the third row has 1 column.

:ROWS is the number of rows. This will normally equal COUNT :COLS.

:CDIST is the distance between columns.

:RDIST is the distance between rows.

:IN denotes indentation characteristics and must be a list. The elements indicate the amount of indentation for each row in relation to the x-position of the left-most motif in the first row. Therefore, there is an indentation value for each

row—except the first. Positive indent numbers mean that the indentation is to the right, and negative ones indicate leftward indenting. For example, suppose :IN is [20 10 -10]. This indicates that the second row is indented 20 units to the right of the start of the first row, the third row is indented 10 units to the right of the first row, and the fourth row 20 units to the left of the first row.

COUNT :IN should normally be equal to (COUNT :ROWS) - 1. Why? If we want no indentation on any row, set indent to [0]. Verify this after you have read through the procedures below. Suppose that we want the odd rows indented by 10 and the even rows by 0? Set :IN to [10 0].

The completed RIFLAG

```
TO RIFLAG :COLS :ROWS :CDIST :RDIST :IN
 ; Indented FLAG procedure with different
 ; number of columns per row possible.
 IF :ROWS < 1 [STOP]
 ROWER :CDIST (FIRST :COLS)
 ; Do a row of the proper column number of images.
 IRSTEP :RDIST :IN
 ; Move down to next row and indent, if necessary.
 RIFLAG (ROT :COLS) (:ROWS-1) :CDIST :RDIST (ROT :IN)
 END

TO ROWER :C :N
 ; To paint a row of :N images separated
 ; by the between-column distance :C.
 IF :N < 1 [STOP]
 RUN :MOTIF
 CSTEP :C
 ROWER :C (:N-1)
END
```

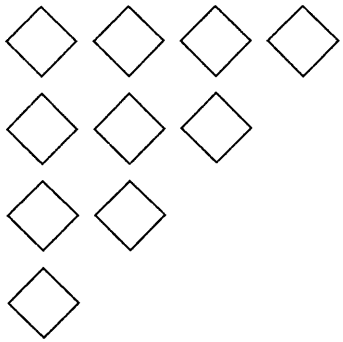
```
TO IRSTEP :R :IN
 PU SETX (FIRST :POINT)  +  (ITEM 1 :IN)
 ; Move back to x-position of the first row's starting
 ; point, then move over the indent amount for the next row.
 RT 180
 FD :R LT 180 PD
END

TO CSTEP :C
 ; This is the same as used for FLAG in Chapter 5.
 PU RT 90
 FD :C
 LT 90 PD
END

TO GO.PT
 PU SETXY (FIRST :POINT) (LAST :POINT)
END
```

Do you understand why ROT and EVAL were used in the RIFLAG procedures? Why must GO.PT be executed before RIFLAG?

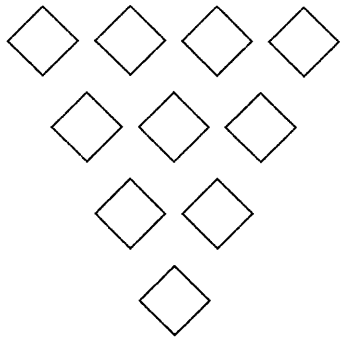
Some RIFLAG examples

The following examples used the motif list MAKE "MOTIF [CNGON 4 20].

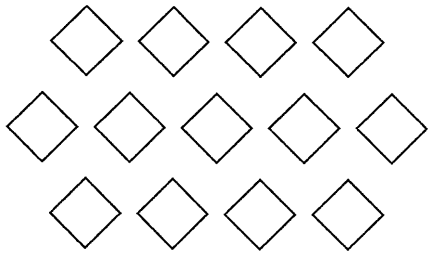
```
GO.PT
RIFLAG  [4 3 2 1]  4  50  50  [0]
```



```
GO.PT
RIFLAG [4 3 2 1]  4  50  50  [25 50 75]
```



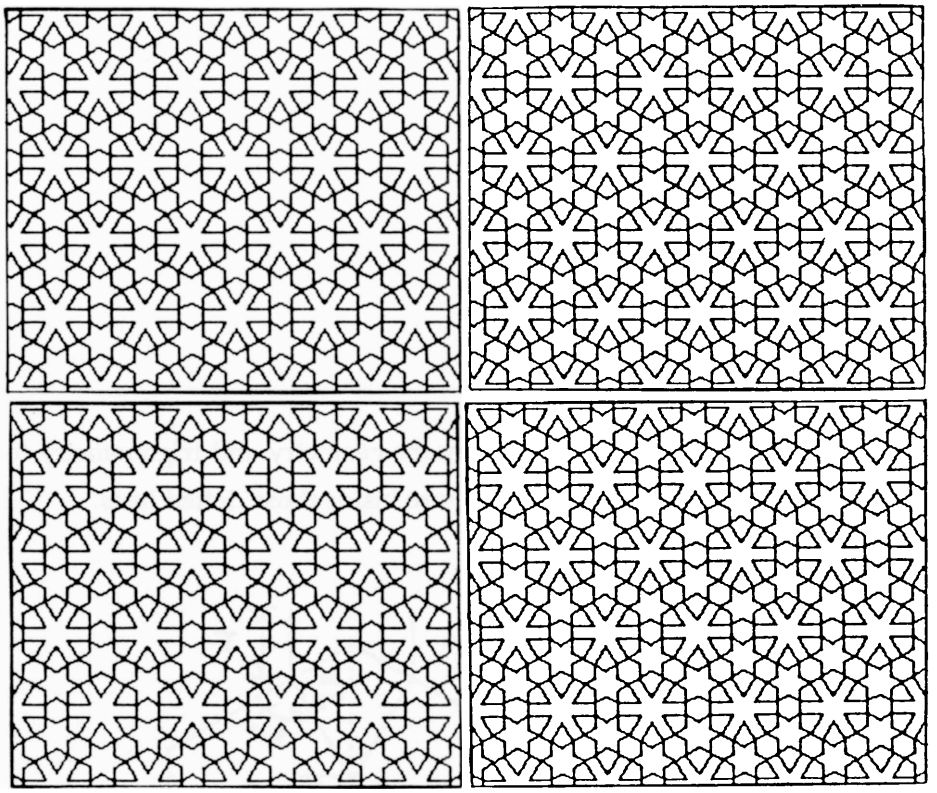
```
GO.PT
RIFLAG [4 5 4]  3  50  50  [-25 0]
```



Starting work on tile design

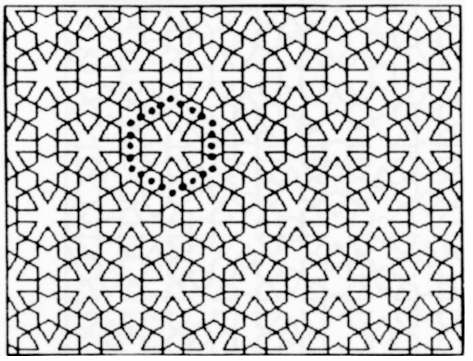
Exercise 5.11 asked you to find an Islamic tile design that appealed to you and to design a suite of Logo procedures that produced designs in the *style* of the design you selected. Your procedures were supposed to be general enough to produce a series of designs, all based on a single theme.

Let me illustrate the approach I would like to see you take by giving you an account of my first Islamic tile exercise. Here is a design that struck my fancy. I saw it in the classic Logo book, *Turtle Geometry*, by Abelson and diSessa. I photocopied the image and pinned it over my desk.

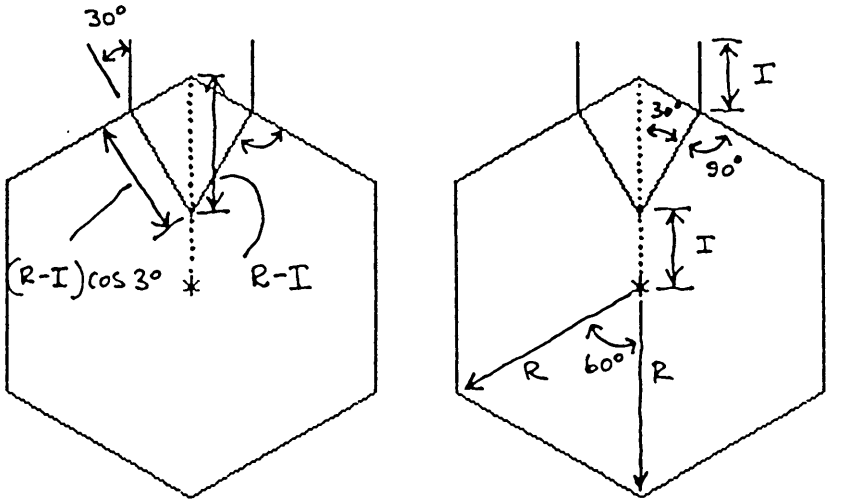


Finding the minimal design elements

The dotted lines enclose what I considered to be the basic design element of the tile pattern: a hexagon on which six V-shaped arms are stuck. I reckoned that if I could write a procedure to draw this basic figure, I could put it into a `:MOTIF` list and then use `RIFLAG` to make a tidy grid of it. The finished rectangular design clearly needs indented rows, so the indenting amounts will have to be calculated.



The next two sketches show the necessary geometry and trigonometry to design the procedures that will draw the figure inside those dotted lines.



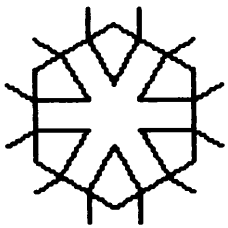
Putting the geometry together

I have defined two characteristics of the basic design, and these characteristics became the two arguments of my procedure. They are `:R`, the radius of the hexagon, and `:I`, one of the dimensions of the V-shaped arms design.

The main procedure is `PIP`. It uses two subsidiary procedures, `BARS` and `ARMS`. Can you "read" them?

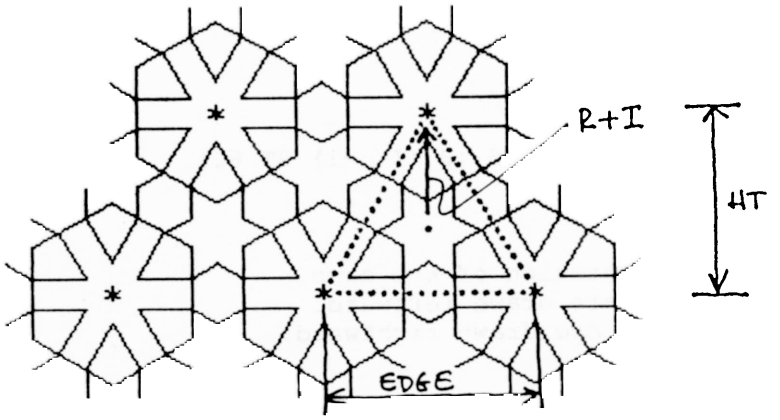
```
TO PIP :R :I
 CNGON 6 :R
 BARS :R :I
END

TO BARS :R :I
 REPEAT 6 [(ARMS :R :I 1) (ARMS :R :I  -1) RT 60]
END

TO ARMS :R :I :F
 ; :F controls the orientation of the arms.
 ; If :F = 1, the arms are drawn leftward.
 ; If :F = -1, the arms are drawn rightward.
 RECORD.POS
 PU FD :I
 LT (30*:F) PD
 FD (:R-:I)*COS 30
 RT (30*:F) FD :I
 RESTORE.POS
END
```

Here is `PIP 60 20`:



Putting `PIP` into a rectangular grid

Before we can make a rectangular grid of `PIP`, we need to know the between-row and between-column amounts. The following diagram suggests how to calculate `:CDIST`, `:RDIST`, and the row indent amounts needed to run `RIFLAG`. Do you see that the dotted triangle is an *equilateral* triangle? The edges are all equal, and the internal angles are each 60 degrees.



Here is a demonstration procedure; use NEST.DEMO to watch the effect changes in :R and :I have on the final design.

```
TO NEST.DEMO :R :I :COLS
 (LOCAL "EDGE "VERT)
 MAKE "EDGE (2*(:R+:I)*SIN 60)
 ; :EDGE is the between-column distance.
 MAKE "VERT :EDGE*COS 30
 ; :VERT is the between-row distance.
 MAKE "MOTIF   [PIP :R :I]
 GO.PT
 RIFLAG :COLS (COUNT :COLS) :EDGE :VERT(LIST (-1*:EDGE/2) 0)
 ; Note how the indent list is assembled.
END
```

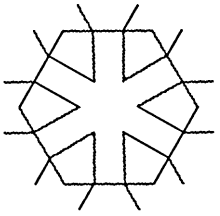
A suite of designs based upon a single tile theme

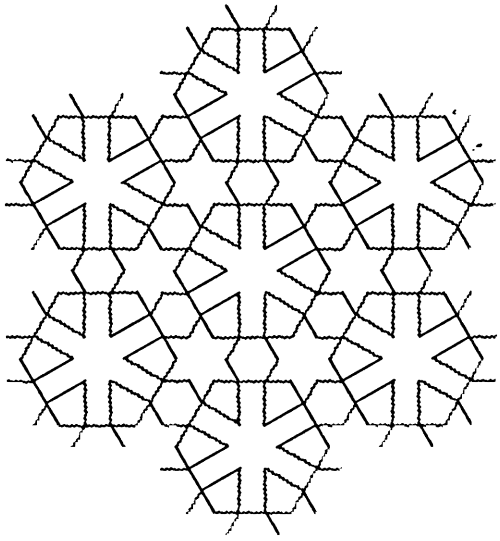
On the following four pages are some experimental results from NEST.DEMO. Each diagram has a different selection of :I and :R values. The smaller figure is drawn with PIP :I :R, and the larger figure with NEST.DEMO :I :R.

The actual :I and :R values used for each diagram are noted. Could you have guessed these numbers? Experiment with some other values, including some negative ones, and then get on with your own design series.

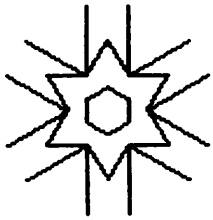
NEST.DEMO designs #1

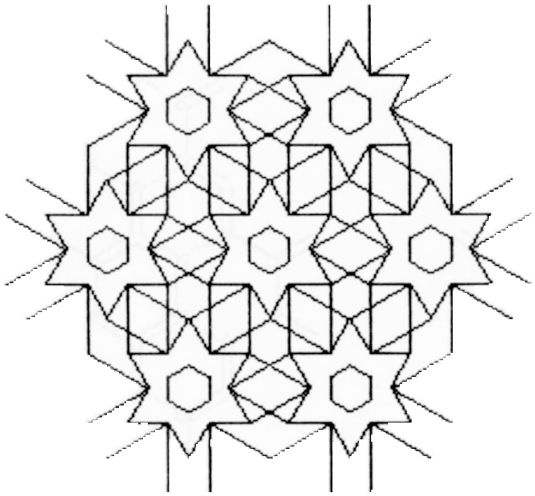
```
PIP 36 12
```





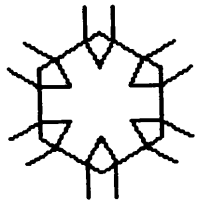
```
PIP 10 30
```

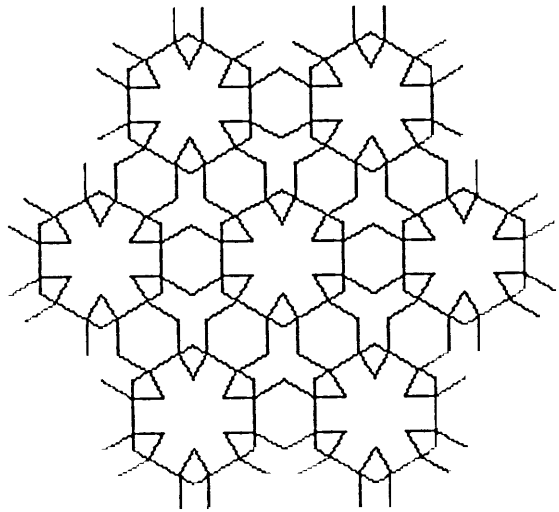




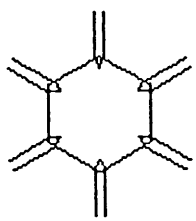
NEST.DEMO designs #2

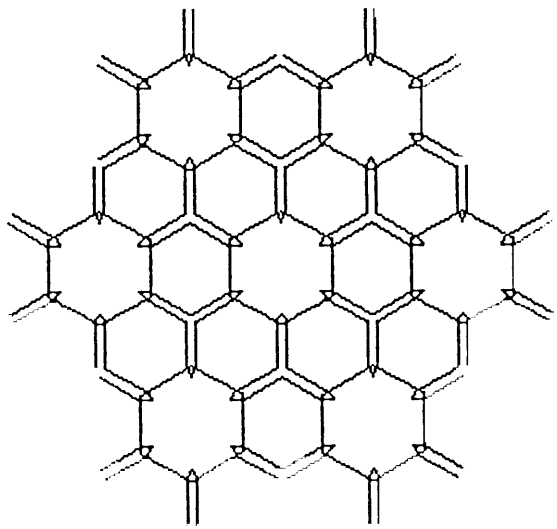
PIP 30 15





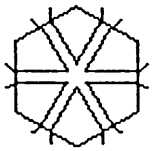
PIP 25 20

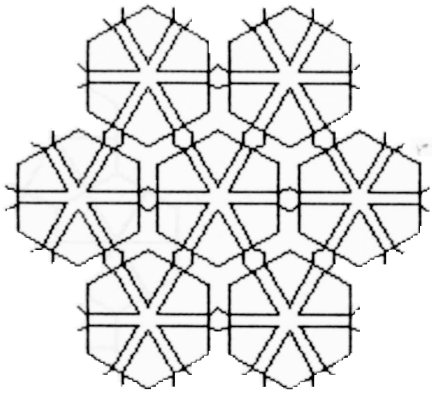




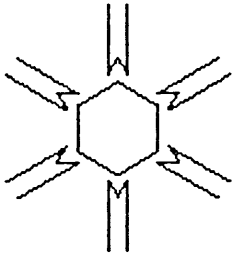
NEST.DEMO designs #3

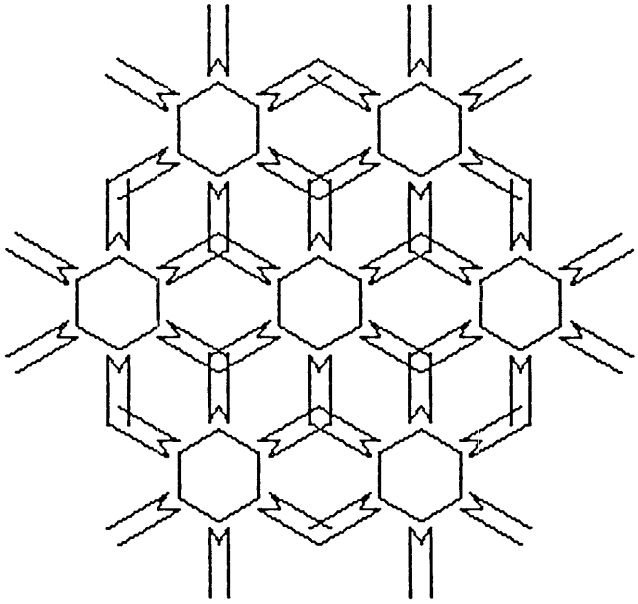
PIP 30 5





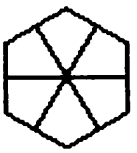
PIP 20 30

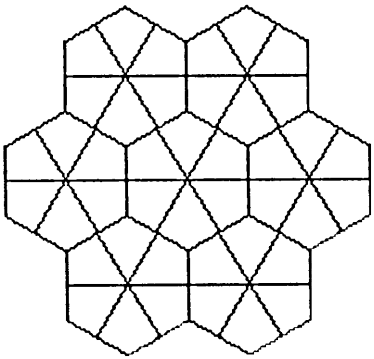


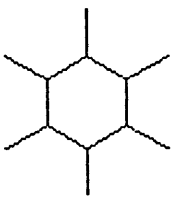


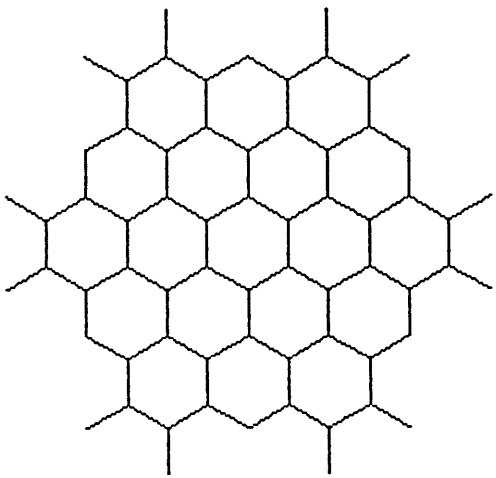
NEST.DEMO designs #4

Can you guess the arguments used for these two designs?





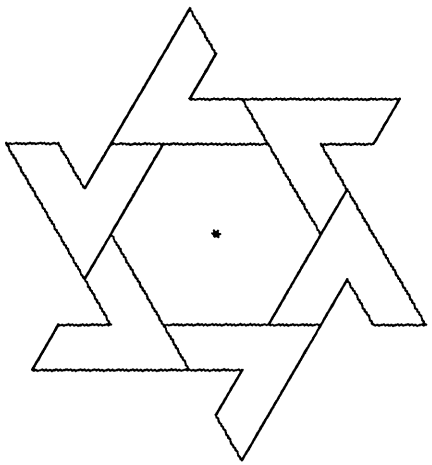




Another tile design exercise

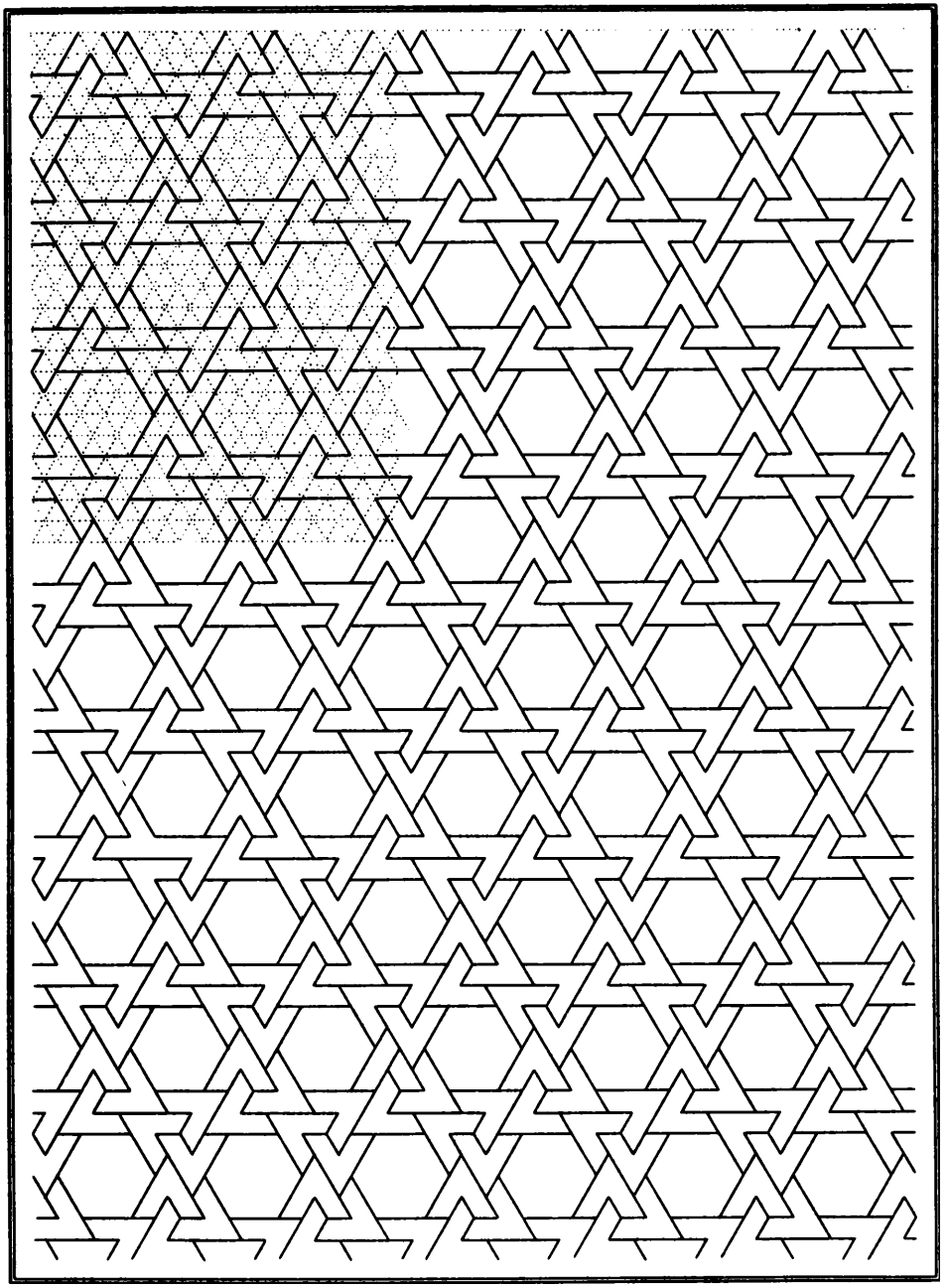
Let's go through another example of dividing an Islamic tile design into its component parts and writing Logo procedures to draw them. Because your visual sense is far more developed now than it was when we started this game, you should be able to decompose designs very quickly and naturally. Test yourself. Look at the design on the next page; can you say *quickly* how to do it? It's easy now, isn't it?

What is the basic design element? Like many other Islamic designs, the basic elements are based on hexagons. Why do you think this is so? Here the hexagon has an odd feature: American Indian teepee shapes are placed on each edge of the hexagon.

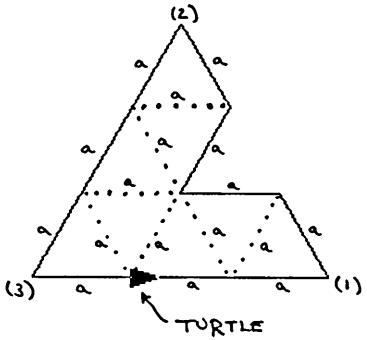


How can we draw this figure? First, let's handle the teepee shape. The outline of the teepee is an equilateral triangle with a triangular bite out of one side. Equilateral triangles keep popping up. In fact, the entire teepee shape can be neatly subdivided into tiny equilateral triangles. Let's call the edge length of these tiny equilateral triangles a. That means that the edge of the teepee shape will be 3a.

Islamic tile design #14



Drawing the teepee shape



The following MAKE statement creates a variable called :TP and makes its value equal to a list that includes all the Logo needed to draw the teepee shape. :TP is just another motif list. The shape will be drawn, starting from the turtle's start position, as shown above, and ending in the same place. So, yes, :TP is a state-transparent list.

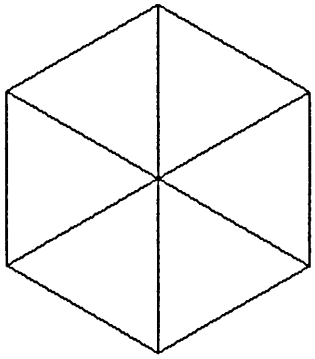
```
MAKE "TP [FD 2*:A -          ; reach point (1)
          LT 120
          FD :A -
          LT 60 -
          FD :A -
          RT120 -
          FD :A -
          LT 60 -
          FD :A -            ; reach point (2)
          LT 120 -
          FD 3*:A -          ; reach point (3)
          LT 120
          FD :A]
```

Before you try to RUN this list, use MAKE to give :A some value. For example, try:

```
MAKE "A 20
RUN :TP
MAKE "A 50
RUN :TP
```

Did you make any mistakes when you typed the value for :TP? If so, you can *edit* what you typed instead of starting all over again. Consult your Logo language manual for the commands to edit names. Editing names is similar to editing procedures.

Now let's put the teepee shape on the edges of a hexagon. Look at the following procedure. TPGON is a very simplified CNGON; it only draws six-sided polygons. It is easy to see that the edge of a hexagon is equal to its radius because hexagons are made from six equilateral triangles.



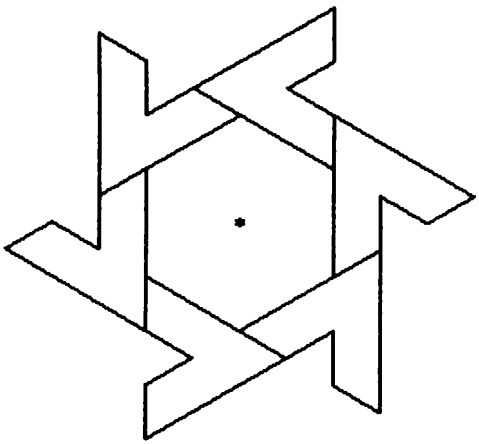
```
TO TPGON :RAD
 PU FD :RAD PD
 RT 120
 REPEAT 6 [FD :RAD RT 60]
 LT 120
 PU BK :RAD
END
```

How can we modify this procedure so that those teepee shapes are placed on the edges of the hexagon? We could express this another way. Remember, back in Chapter 3, when we spoke about changing the *quality of the edges* of polygons? There we changed the straight line quality of a polygon's edge into a kinked star edge. Later in the chapter, we installed a fractal edge. Now we want to change the straight line quality of hexagon edges into "teepee edge quality."

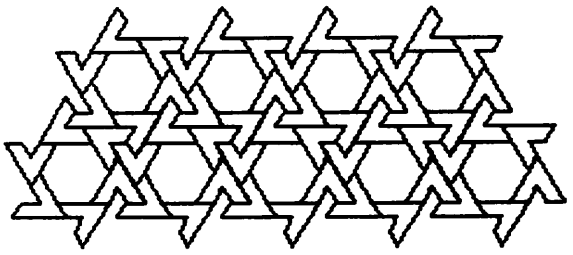
Here it is:

```
TO TPGON :RAD
 PU FD :RAD PD
 RT 120
 LOCAL "A MAKE "A :RAD/2
 REPEAT 6 [RUN :TP FD :RAD RT 60]
 LT 120
 PU BK :RAD PD
END
```

And here is an example of its visual output:

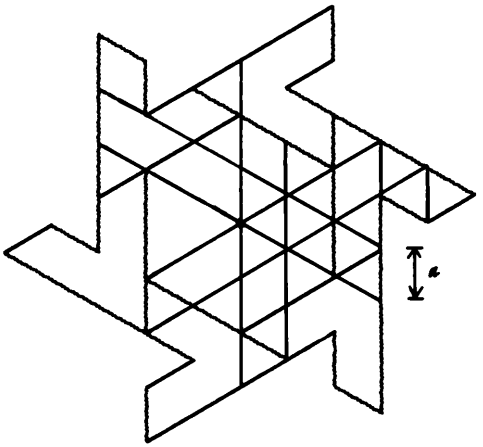


What next? We can put TPGON into an RIFLAG. TPGON goes into the :MOTIF list, but then we must calculate the values for the :RDIST, :CDIST, and :IN arguments. What we *want* to end up drawing is shown at the top of the next page. But before we discuss how to carry out these calculations, let's summarize what we have done so far. After we summarize, we will sermonize, offering a few rules on breaking down designs into their basic components. We might be able to apply these rules to other Islamic designs. In preparation for all this, go back and look again at the teepee grid.



Breaking down complex designs

When you first looked at this design, it probably looked complex; the basic design elements were not initially obvious. But a closer look uncovered the basic component: six teepee shapes placed on the edges of a hexagon. Deeper inspection showed that both the hexagon and the teepee shapes could be broken down into small equilateral triangles. The edge dimension of these small triangles conveniently became the basic dimension of the entire design, and we labeled this edge length :A.



We defined the teepee shape with a Logo list; this list was incorporated into a procedure that drew a hexagonal composite, or swirl, of teepee shapes;

and finally, we wanted to make the name of this swirl-drawing procedure the value of the variable :MOTIF. We also planned to use RIFLAG to draw teepee swirl grids.

A small review and a few lessons

Note the style of problem solving that we followed above. A complex visual problem was broken down into smaller design elements. Each was structured with either a Logo procedure or a Logo list. The final design was constructed by putting the individual Logo pieces together.

Note the usefulness of the concept of *naming* in all this. Once we know how to do something in Logo—like drawing the shape of a teepee using a list of Logo drawing commands—we give the method a short name, like :TP. The name of the method now represents all the features of the method but is short and tidy. Once we name a method, we can forget how its inner mechanism operates. We can concentrate our energies on the next problem.

Now for the *rules* for finding the smallest design elements of a complex figure. Keep your eye open for simple polygonal structures. Triangles/hexagons and squares/octagons are ubiquitous because they fit, or "tile," together so well. Look carefully, though. You may not see the basic polygons because their edge "qualities" have been altered.

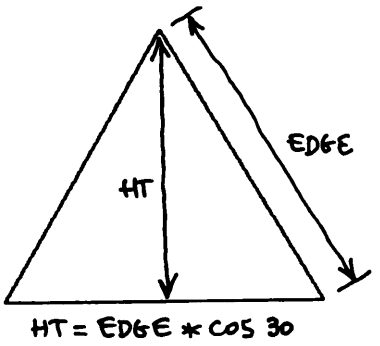
Finally, let's work out the necessary row and column dimensions of these teepee swirl grids.

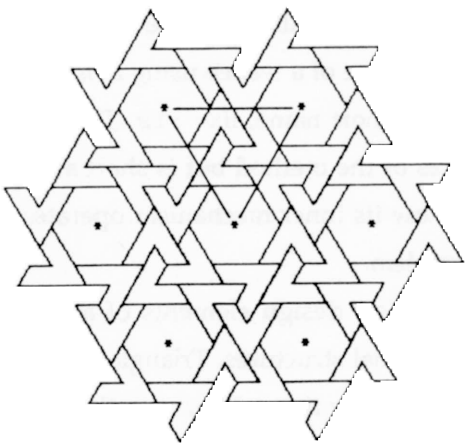
The three diagrams on the following page illustrate the calculations needed. We are again manipulating the elements of equilateral triangles.

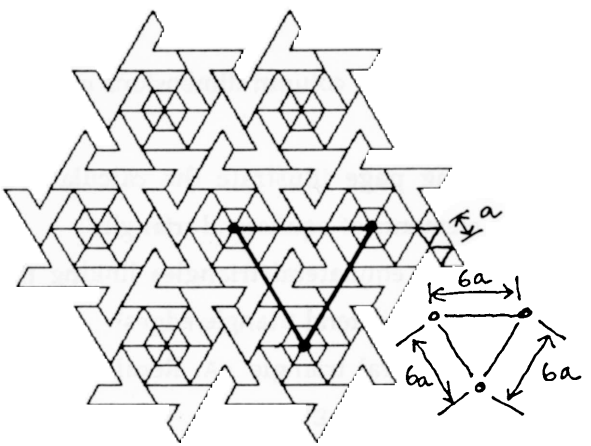
Do you see that the edge of the large equilateral triangles linking the centers of the teepee swirls equals six small equilateral triangle edges?

We called the edge of the small equilateral triangle :A. Recall that the height of any equilateral triangle is equal to either :EDGE*SIN 60 or :EDGE* COS 30. Why are these two forms equivalent?

Grid placement calculations for teepee swirls







Let's use this geometry to write a demonstration procedure that will fit the teepee figures together. This routine should work for any size teepees.

```
TO TP.DEMO :SIZE :COLS
   :SIZE is the radius of the hexagon on which the teepees
 ; will be placed. Remember that :A is :SIZE/2. :COLS must
 ; be a list whose elements define the columns per row.
 (LOCAL "EDGE  "VERT)
 MAKE "EDGE 6*:SIZE/2
 ; :EDGE is the between-column distance.
 MAKE "VERT :EDGE*COS 30
 ; :VERT is the between-row distance.
 MAKE "MOTIF   [LT 30 TPGON :SIZE RT 30]
 GO.PT
 RIFLAG :COLS (COUNT :COLS) :EDGE :VERT(LIST (-1*:EDGE/2) 0)
 ; Note how the indent list is assembled.
END
```

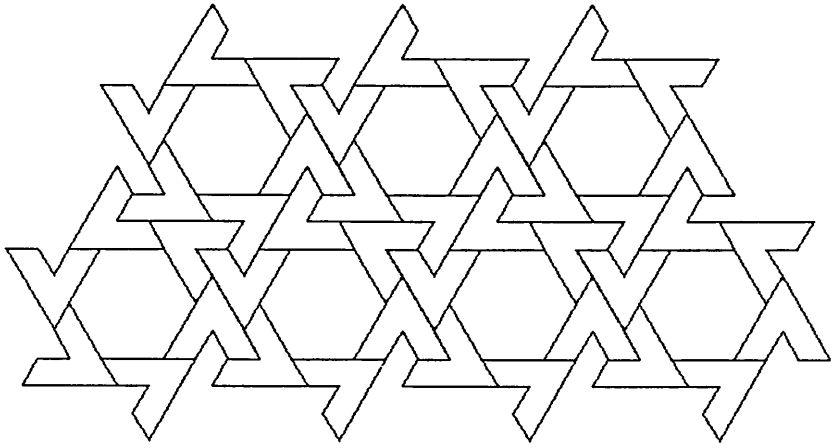
Compare TP.DEMO with NEST.DEMO. Why are they so alike? Because both demonstration procedures place design elements at the vertices of equilateral triangles.

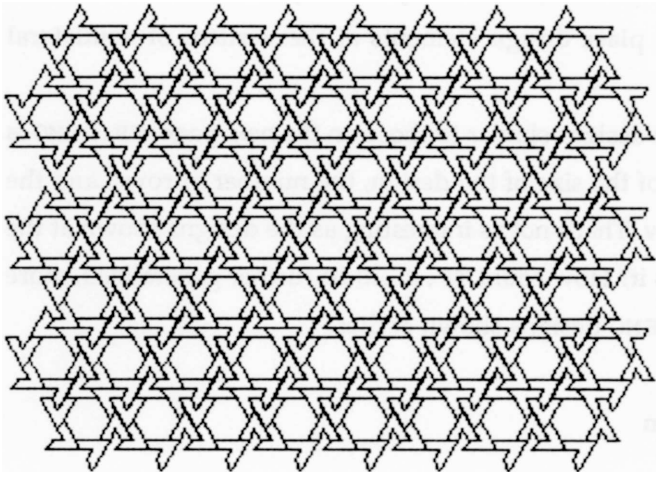
TP.DEMO can produce designs similar to those on the next page. TP.DEMO is generalized only in terms of the size of the design, the number of rows, and the number of columns per row. That's not as interesting as the designs shown at the beginning of the chapter, is it? How could TP.DEMO be further generalized, more in the spirit of the NEST.DEMO designs shown earlier?

A star and saw blade design

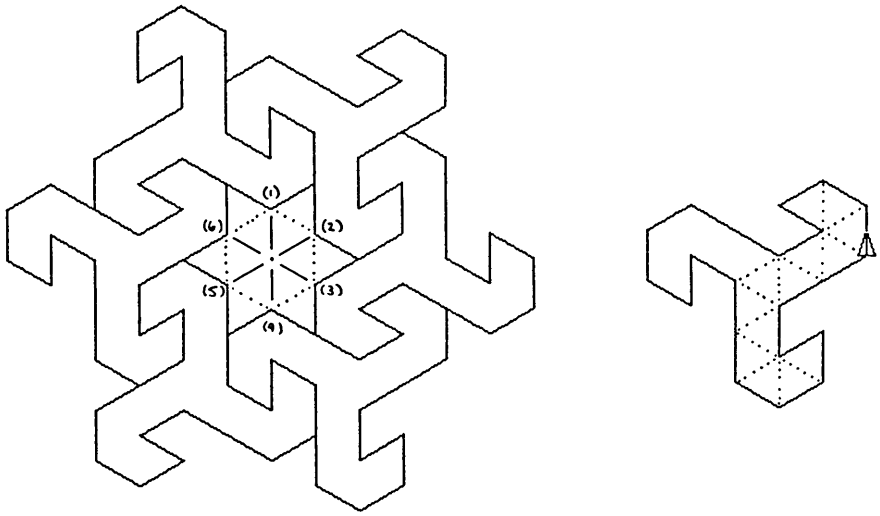
Let's try your eye on Islamic Tile Design #4 from Exercise 5.11. Note several things. First, note where the turtle starts drawing the saw blade. This is not arbitrary. The turtle begins at the vertex in the saw blade that will touch each of the vertices of the underlying hexagon. (Go back and look at the STARGON discussion in Chapter 4 where we talked about polygon edge qualities and the rules that govern the replacement of one kind of edge with another.) Look at the

Teepee swirl grids





sketch below. The dotted figure is this underlying hexagon. Remember that all hexagons are composed of little triangles. Let's have the turtle go to each of the hexagon vertices in turn; we can use a CNGON 6 :RAD to do that. Therefore, the starting place in the saw blade list must correspond to where the saw blade figure touches the vertices of the small, dotted hexagon.




```
MAKE "SAW [REPEAT 3 [REPEAT 3 [FD :A LT 60] -
           LT 60 FD :A RT 120 FD :A RT 60 -
           FD 2*:A LT 60] ]
```

The second thing to note about the list :SAW is that it is state transparent. Why is this necessary?

Finally, note the uses made of the nested REPEAT commands within the list. The saw blade figure is actually a composite of a simpler form that is repeated three times. Do you see this hook form?

You might test yourself by writing your own alternative list structure for the saw blade figure. Start your list at a different point or draw the figure in a clockwise rather than a counterclockwise direction. Remember, though, that the start of the :SAW list must correspond to the hexagon drawing procedure. Remember that the :SAW shape is being placed on the edge of a hexagon.

Putting the saw blade into a hexagon swirl

The following procedure, SAWGON, is similar to TIPGON from Chapter 4. Rather than retracing the shape of the underlying hexagon, placing the sawblades on each edge (which is what TPGON did), SAWGON begins at the hexagon's center and moves out to each vertex in turn. Once there it runs the SAW list and then returns to the center. It then turns to face the next vertex and moves out to it. This is repeated six times.

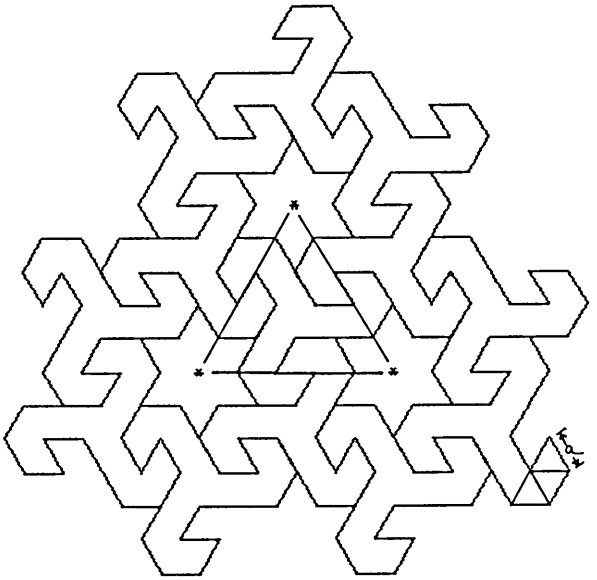
```
TO SAWGON :A
 ; To place six sawblades on the edges of a hexagon.
 ; Note that the turtle must turn right 60 degrees
 ; before RUNning the list :SAW. After RUNning :SAW
 ; the turtle turns left 60 degrees. The radius of the
 ; underlying hexagon is :A. Why?
 REPEAT 6 [PU FD :A PD -
           RT 60 RUN :SAW LT 60 -
           PU BK :A -
           RT 60]
 PD
END
```

Putting SAWGON into RIFLAG

We must now calculate the distance between rows and columns of these SAWGONs so that they will fit neatly together. We want the hooks of the design to close properly—one hook just inside another. The design will have indented rows, so this amount must also be calculated.

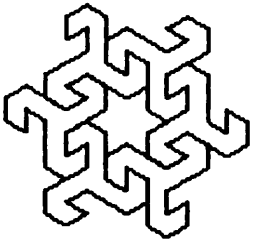
Look at the design on the next page and compare it with the previous teepee grid calculations. We are again faced with large and small equilateral triangles.

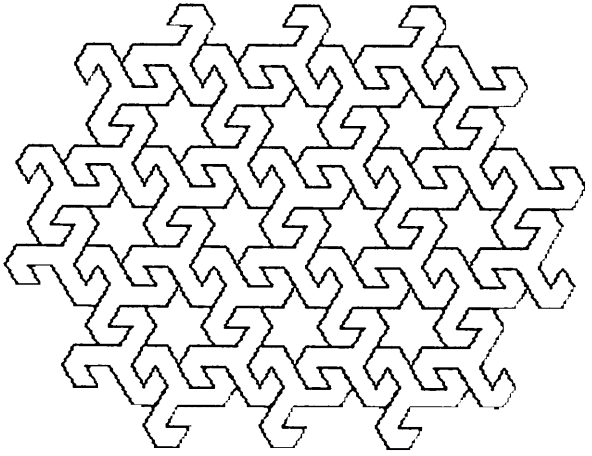
The SAWGON swirls are centered on the vertices of equilateral triangles. This geometry has been incorporated in the demonstration procedure SAW.DEMO. It is just like NEST.DEMO and TP.DEMO.



```
TO SAW.DEMO :SIZE :COLS
 ; :SIZE is the dimension of the small
 ; equilateral triangle a.
 (LOCAL "EDGE "VERT)
 MAKE "EDGE 5*:SIZE
 ; :EDGE is the between-column distance.
 MAKE "VERT :EDGE*COS 30
 ; :VERT is the between-row distance.
 MAKE "MOTIF  [LT 30 SAWGON :SIZE RT 30]
 ; Why the 30 degrees tilting of SAWGON?
 GO.PT
 RIFLAG :COLS (COUNT :COLS) :EDGE :VERT(LIST (-1*:EDGE/2) 0)
 ; Note how the indent list is assembled.
END
```

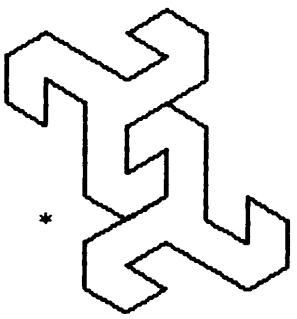
Composite designs that hook nicely together

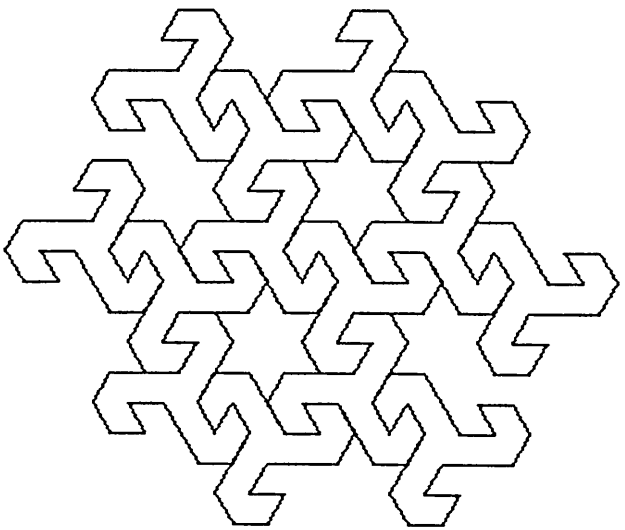




Simplifying the SAWGON swirl

You may have noticed that we should modify the SAWGON procedure since it draws too many SAW shapes. Several SAWs are repeated, one on top of another. In fact, SAWGON needs to draw only two saw blades rather than six for the composite design to mesh. The two diagrams below should convince you.

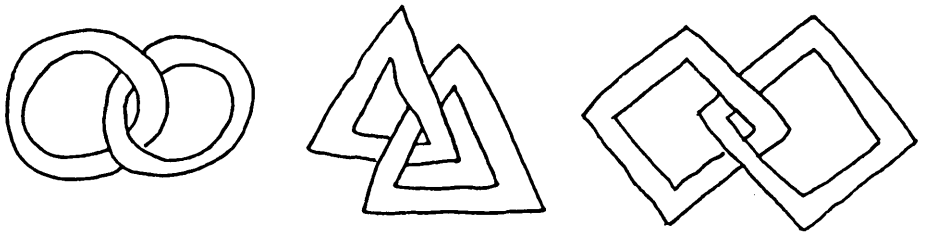




Overlapping designs

Go back and look at Islamic tile designs 11-13 at the end of Chapter 5. Notice that these designs look as if they were woven from a pliable material. Part of the design passes behind other parts, and this woven quality seems to follow some consistent design rules. Can we simulate these designs with the techniques we have developed so far?

To start your thinking, make some sketches of a few interlocking rings. Let's actually look at interlocking polygon rings. Here are a few sketches of interlocking circle, triangle, and square rings:

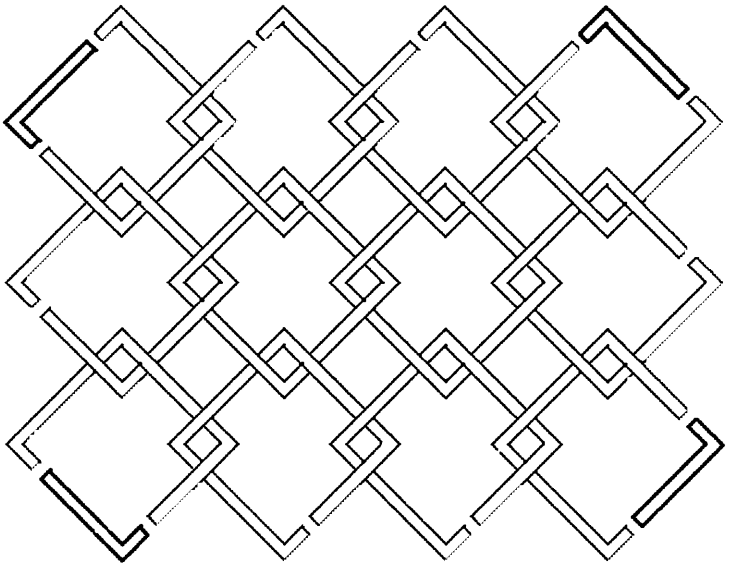


Suppose we want to draw these designs using Logo. Is it possible to break the design down into smaller component parts that can be defined by lists or small procedures? Are these basic design elements placed around some invisible shape—as the teepees and saw blades were placed around an invisible hexagon?

Perhaps the following exercise will give you some ideas.

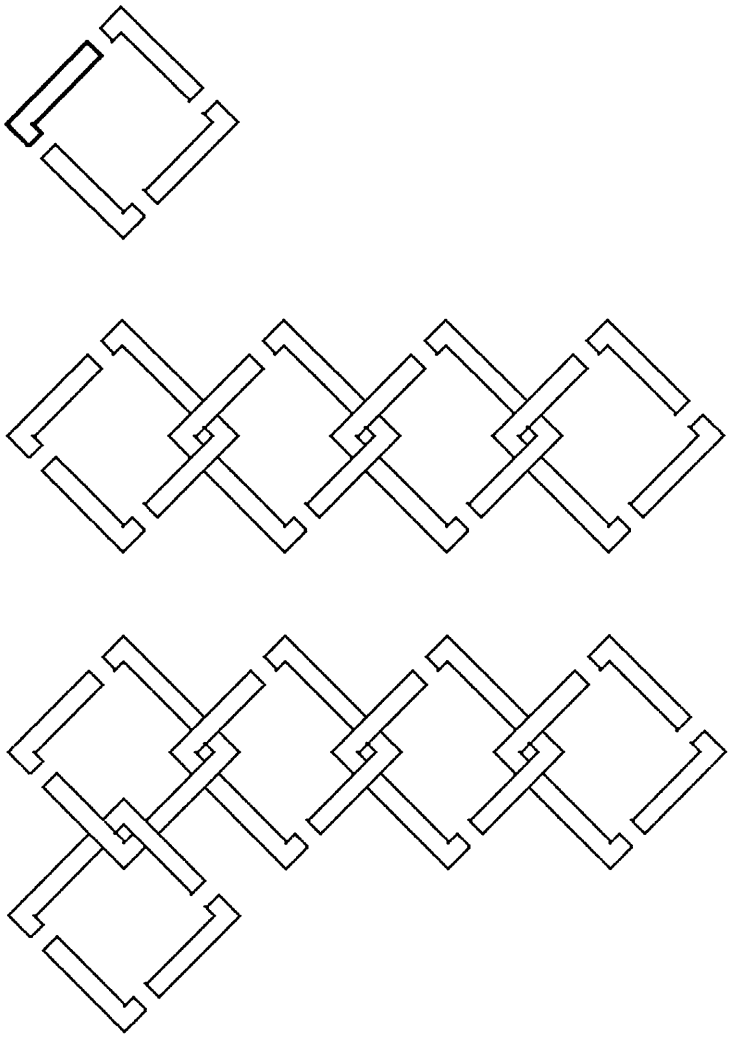
A grid of interlocking square rings

Look closely at the grid below. Would the component design elements be different if the grid had only a single row? Why?



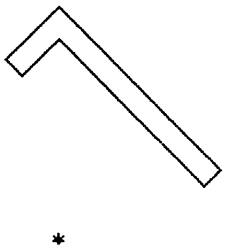
An animated assembly of square rings into a grid

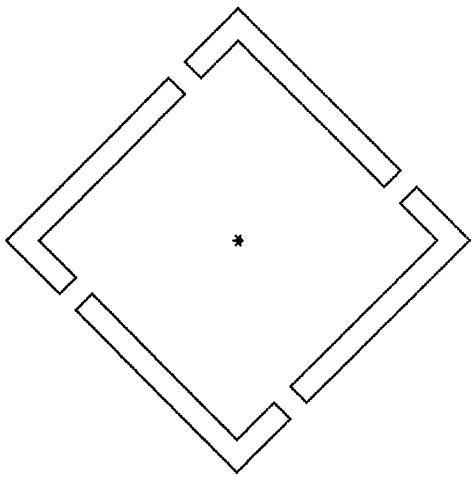
Now look hard at the following.



The basic design components

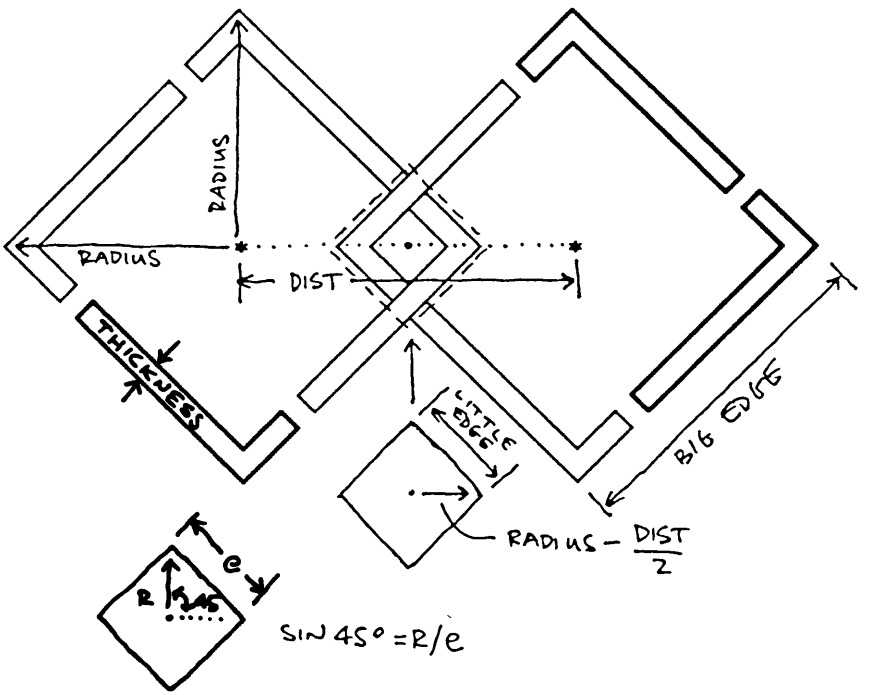
The following two diagrams illustrate what we want. The first shows the basic element and the second shows the element placed around a square.





Overlapping square rings calculations

The characteristics that will be needed to draw the basic design element are: the radius of the square rings, the thickness of the rings, and the distance between the rows and columns of these rings in a final ring grid. Let's start with equal row and column distances. The following diagrams illustrate one way of modeling this ring problem. It isn't the only way of doing it. Perhaps you would be happier with another method. That's OK. As long as you create "ringy" shapes. *Build a model of these shapes in a manner that is most natural to you.*



So for elements above

BIG EDGE (:BE) = RADIUS / SIN 45°

LITTLE EDGE (:LE) = (RADIUS - DIST/2) / SIN 45°

:TH = THICKNESS

Overlapping square rings procedures

The procedures below follow the diagrams on the previous page. WING draws the individual component, and RINGS places this component around an invisible square. You should see the similarity between RINGS and TPGON. RINGS places an element around a square, while TPGON places an element around a hexagon.

```
TO WING :BE :LE :TH
 ; State-transparent element that is
 ; used by RINGS procedure.
 RT 135
 FD (:BE-:LE) RT 90 FD :TH RT 90
 FD (:BE-:LE-:TH) lt 90
 FD (:LE-2*:TH) RT 90 FD :TH
 RT 90 FD (:LE-:TH)
 LT 45
END

TO RINGS :RAD :TH :DIST
 ; Interlocking square rings. :RAD is radius of squares,
 ; :TH the thickness of the rings, and :DIST the
 ; horizontal and vertical distance between the
 ; centers of square rings in a grid.
 (LOCAL "BE  "LE)
 MAKE "BE :RAD/(SIN 45)
 MAKE "LE (:RAD-(:DIST/2))/(SIN 45)
 REPEAT 4 [PU FD :RAD PD -
           WING :BE :LE :TH -
           PU BK :RAD RT 90]
 PD
END

TO RING.DEMO :RAD :TH :DIST :COLS
 MAKE "MOTIF [RINGS :RAD :TH :DIST]
 GO.PT
 RIFLAG :COLS (COUNT :COLS) :DIST :DIST ( [0] )
END
```

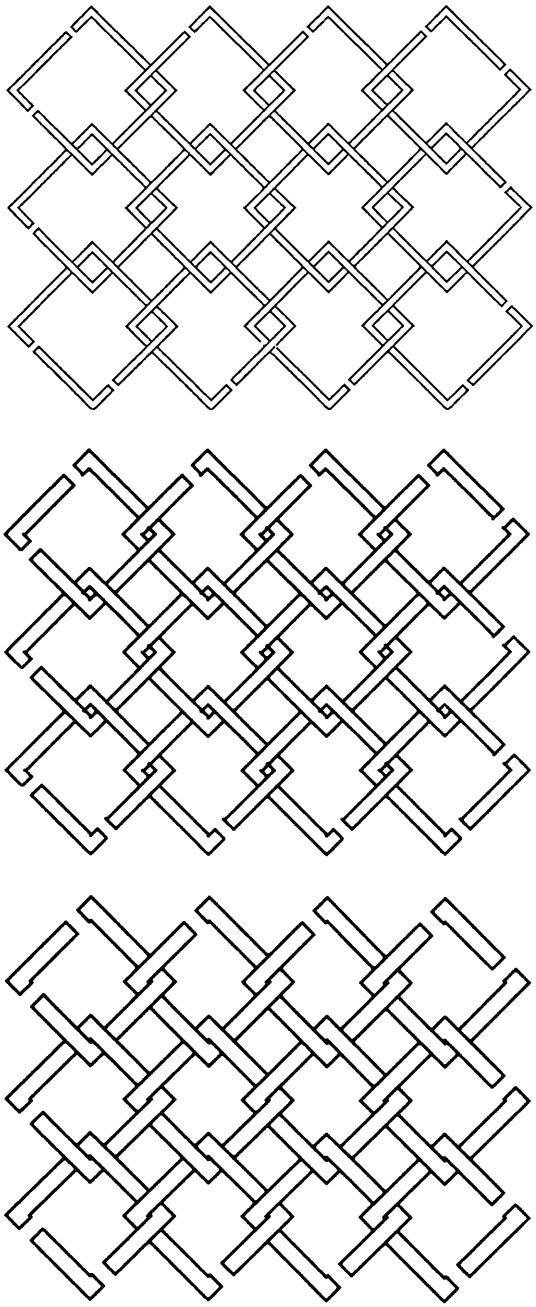
An alternative structuring of WING and RINGS

You probably compared the design element WING with SAW and TP. The two latter elements were designed as lists, while WING is structured as a procedure. You will come to realize that lists and procedures are often interchangeable in Logo. Below is the list alternative to the procedures on the previous page. Take some time to think about the differences between the two approaches. Do you find one more aesthetic than the other? Why?

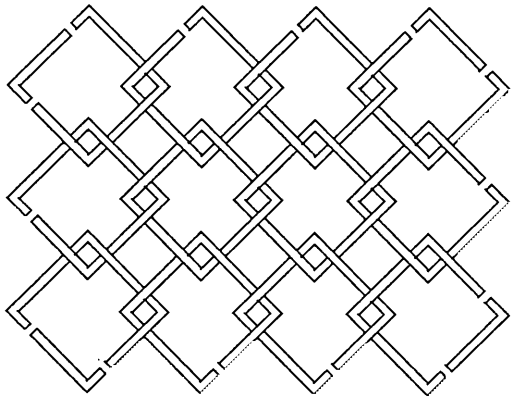
```
MAKE "WING [RT 135 -
            FD (:BE-:LE) RT 90 FD :TH RT 90 -
            FD (:BE-:LE-:TH) LT 90 -
            FD (:LE-2*:TH) RT 90 FD :TH -
            RT 90 FD (:LE-:TH) -
            LT 45]

TO RINGS :RAD :TH :DIST
 ; Interlocking square rings. :RAD is radius of squares,
 ; :TH the thickness of the rings, and :DIST the
 ; horizontal and vertical distance between the
 ; centers of square rings in a grid.
 (LOCAL "BE "LE)
 MAKE "BE :RAD/(SIN 45)
 MAKE "LE (:RAD-(:DIST/2))/(SIN 45)
 REPEAT 4 [PU FD :RAD PD -
           RUN :WING -
           PU BK :RAD RT 90]
END
```

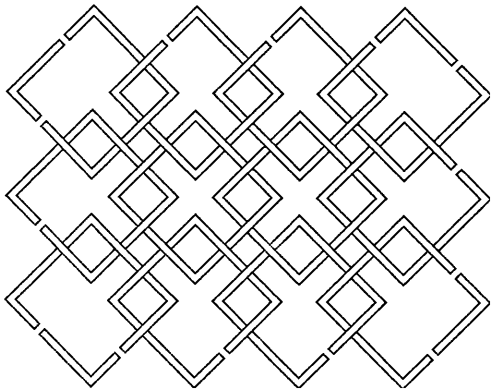
The procedure RING.DEMO is the same for both of the approaches.

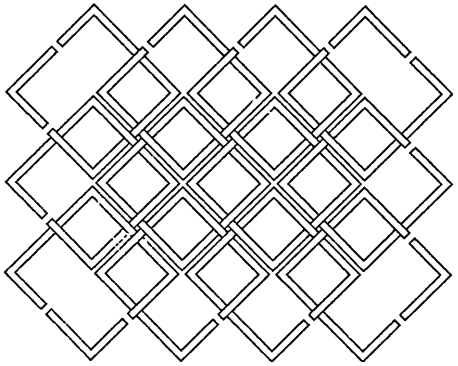
Three ring grids of different thicknesses, with all other characteristics fixed



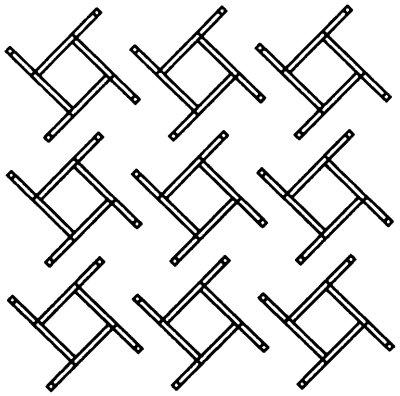
Three ring grids with different separations, with all other characteristics fixed

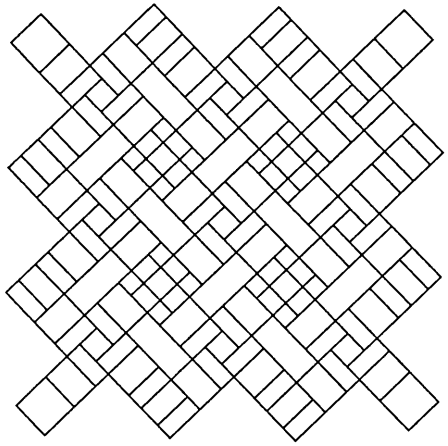


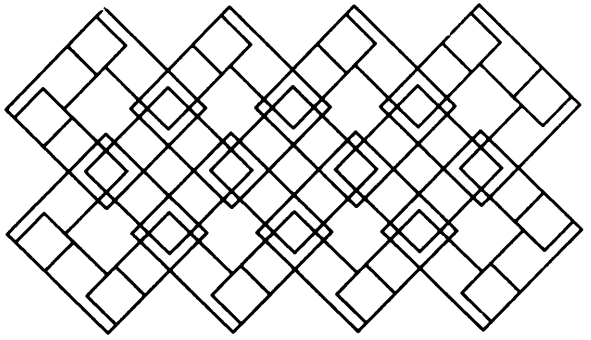




Three ring grids with "extreme value" arguments







Exercises

Exercise 6.1

The TP and saw blade designs were composed of tiny equilateral triangles. Can you design a single Logo procedure that could produce either of these designs? You would have to find characteristics that describe the placement of equilateral triangles in a basic design element. Then, by changing the values of these characteristics, your procedures could draw either design. And they could also draw a series of alternative designs, all based on triangles and all fitting nicely together in a grid.

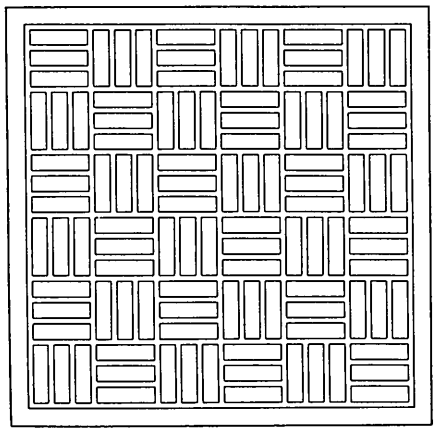
Start by designing a procedure to draw a large rectangular grid of small, interlocking, equilateral triangles. Print this grid onto a good grade of heavy paper. You may want to consider blowing up your triangle-grid image with an enlarging photocopy machine. Find a pad of translucent tracing paper and place a sheet over the triangular grid. Trace out new tile designs on the translucent paper, structuring the components with tiny triangle properties. Translate your designs into procedures.

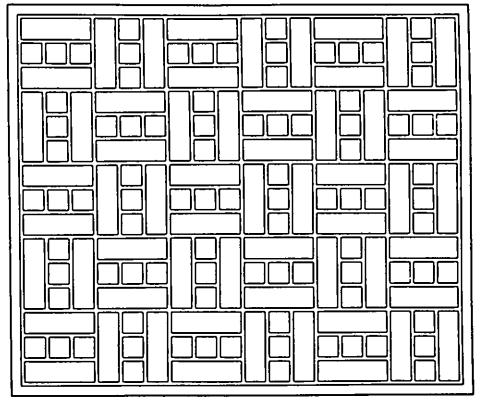
Carry out the same experiment with other shapes that tile together. Which polygons do?

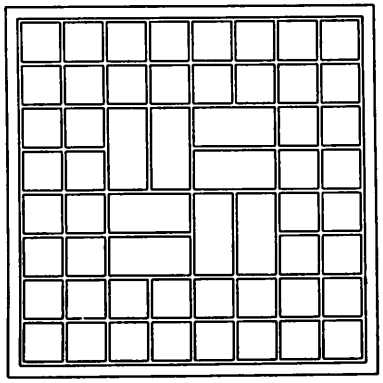
Exercise 6.2

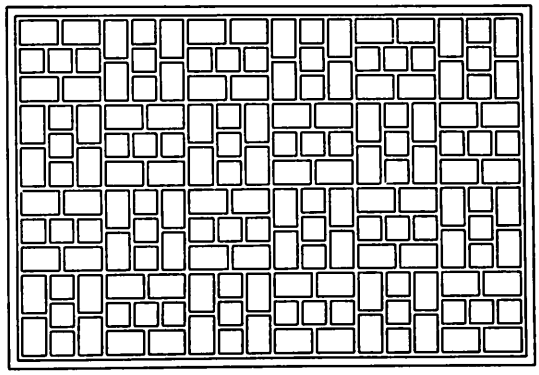
I recently discovered Daniel Sheets Dye's book, *Chinese Lattice Designs*, republished by Dover in 1974. This book has 1200 illustrations; I offer you only several here. Can you use the design methods of this chapter to analyze these images? Can you design procedures that are generalized enough so that they can produce several of the designs shown? See if you can combine some of the Islamic tile designs we have studied with the lattice designs below. Can you express in words the difference in feeling between Islamic designs and those shown below?

Lattices for Exercise 6.2

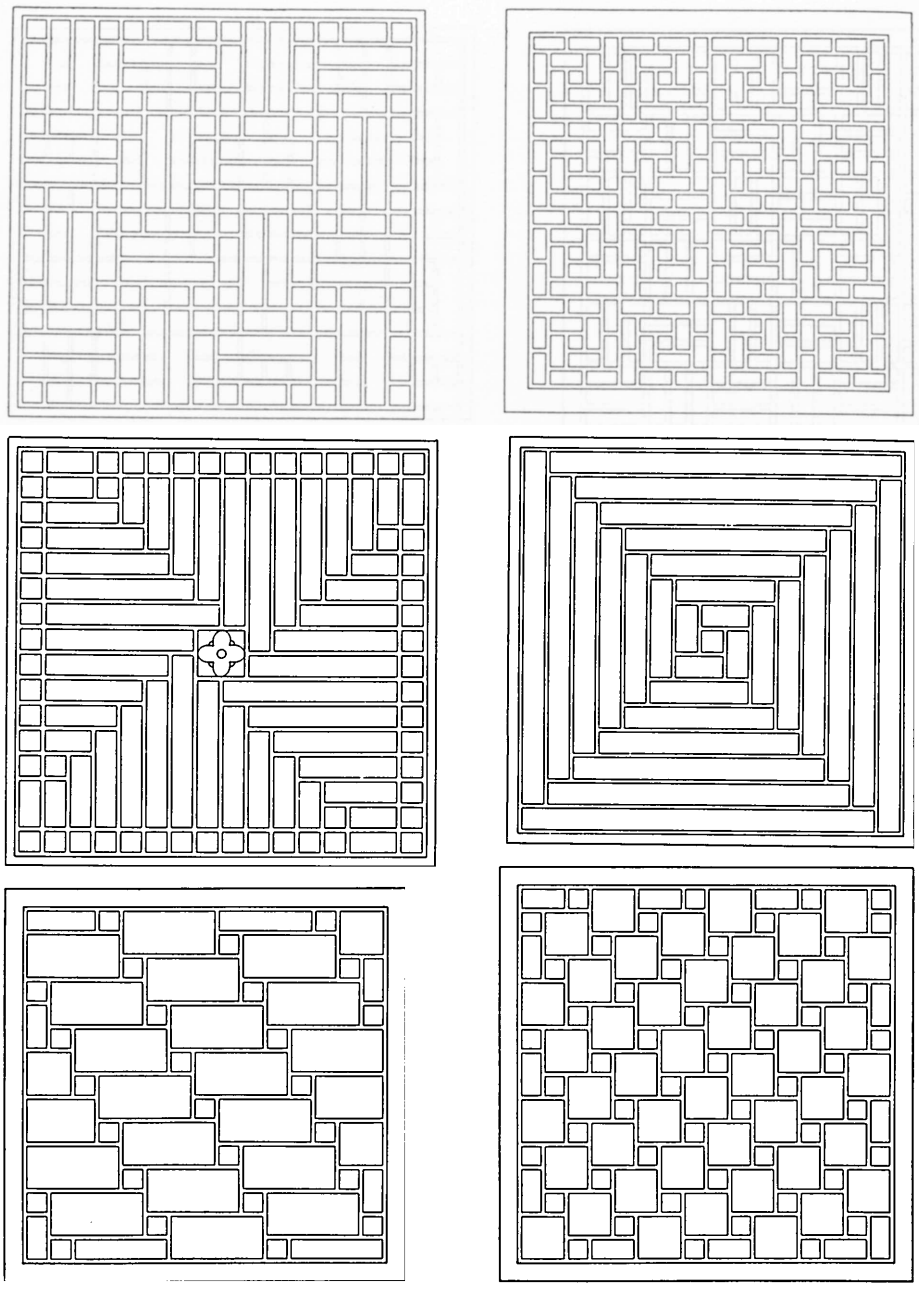








More lattices for Exercise 6.2



Exercise 6.3

Every city has oriental rug merchants. Visit one of these vendors, but remember to take your sketchbook along with you. Record several rug designs that strike your fancy. Explore these designs with a suite of Logo procedures. Hand color the results and place them in your notebook.

Exercise 6.4

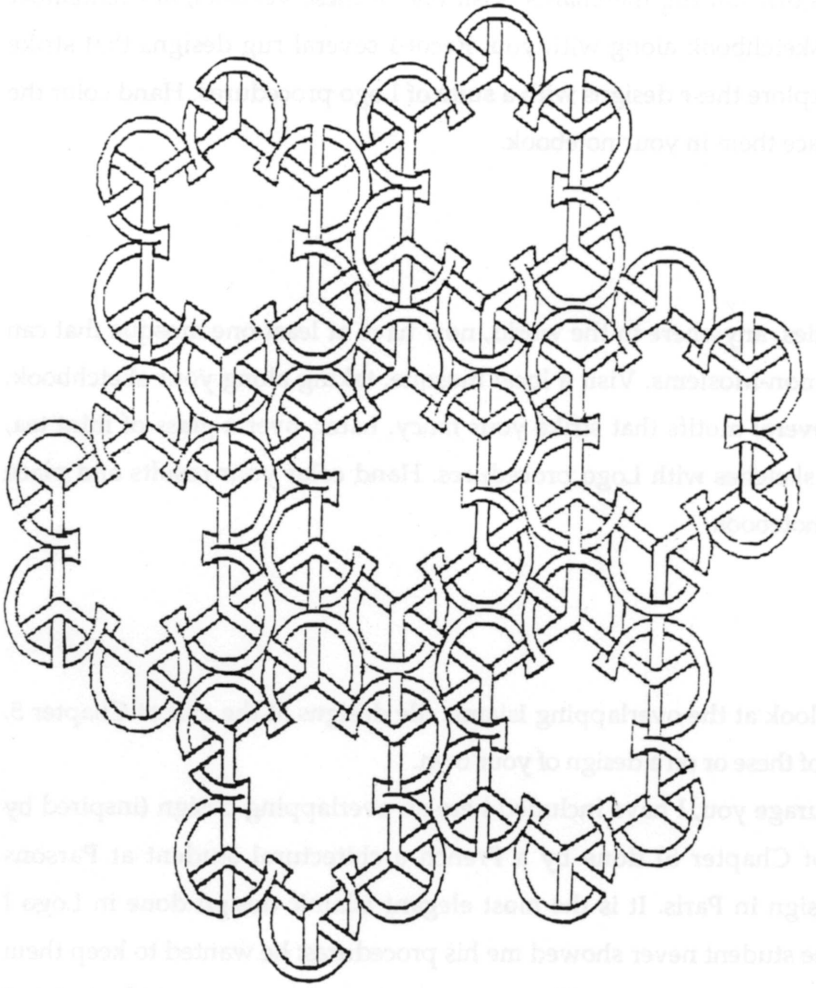
Most large cities, anywhere in the world, now have at least one mosque that can be visited by non-Moslems. Visit a local mosque, taking along your sketchbook, and record several motifs that strike your fancy. Later, over a glass of mint tea, explore your sketches with Logo procedures. Hand color your results and place them in your notebook.

Exercise 6.5

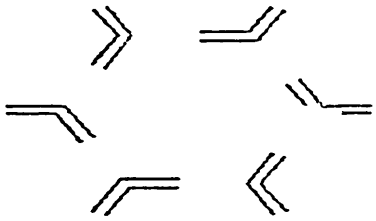
Go back and look at the overlapping Islamic tile designs at the end of Chapter 5. Work on one of these or on a design of your own.

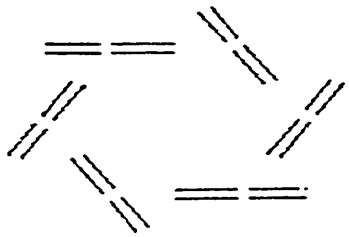
To encourage you, I have included here an overlapping design (inspired by design #12 of Chapter 5) done by a French architectural student at Parsons School of Design in Paris. It is the most elegant Islamic design done in Logo I have seen. The student never showed me his procedures; he wanted to keep them "secret." He did show me a few illustrations of the component parts he isolated in his design. I have also included his visual investigation of the effect of varying the width of these basic design elements.

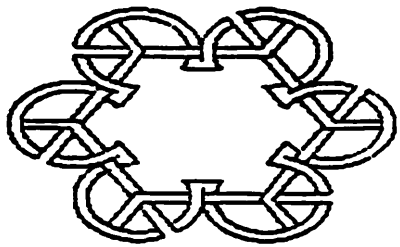
Student-designed overlapping Islamic design

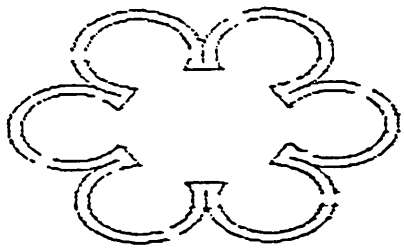


Component parts of the student design

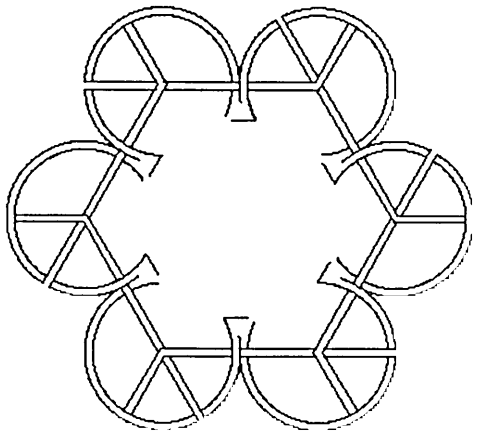


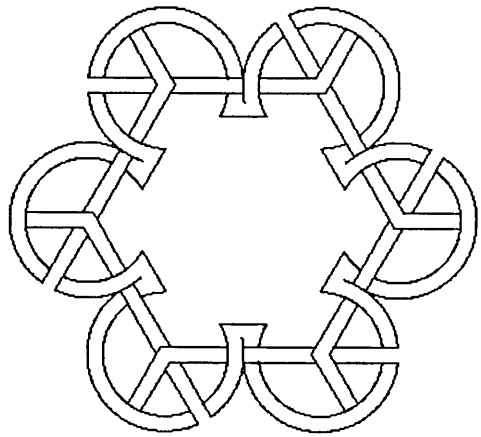


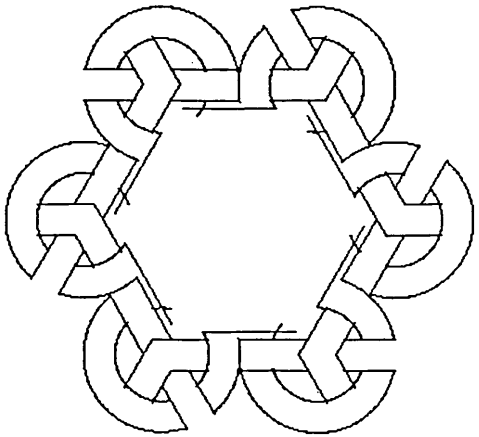




Visual exploration of different design element thicknesses

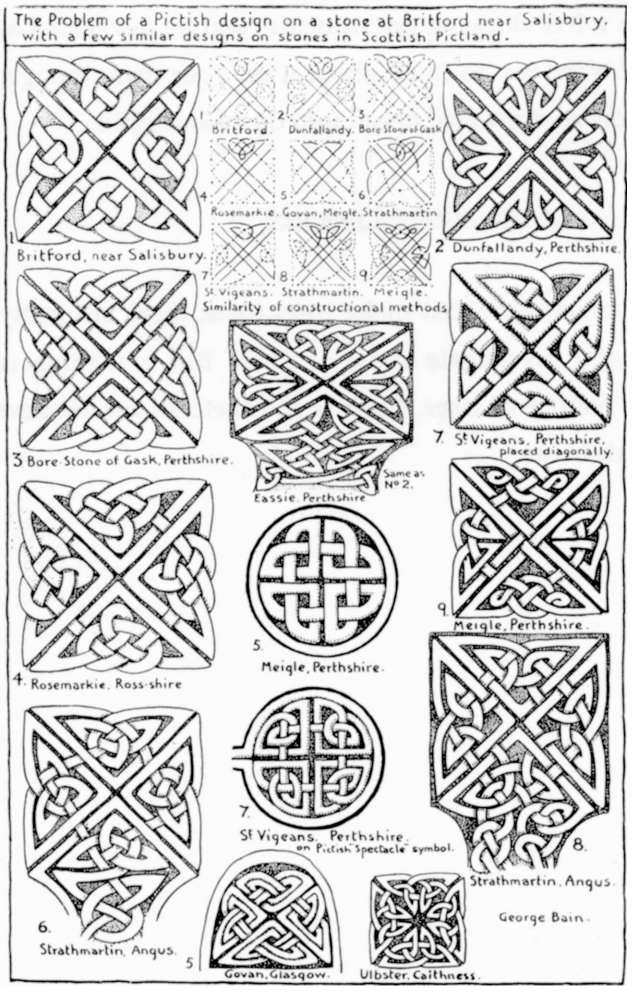






Exercise 6.6

Celtic art used overlapping patterns extensively. The forms are based, obviously, on the direct observation of knotted rope and other materials. Try the empirical approach. Get some rope or heavy twine and tie a few knots. Sketch what you see and try to reproduce the results with Logo procedures. I include some Celtic examples to get you started. This illustration is taken from *Celtic Art: the methods of construction* by George Bain (Dover, 1973).



Exercise 6.7

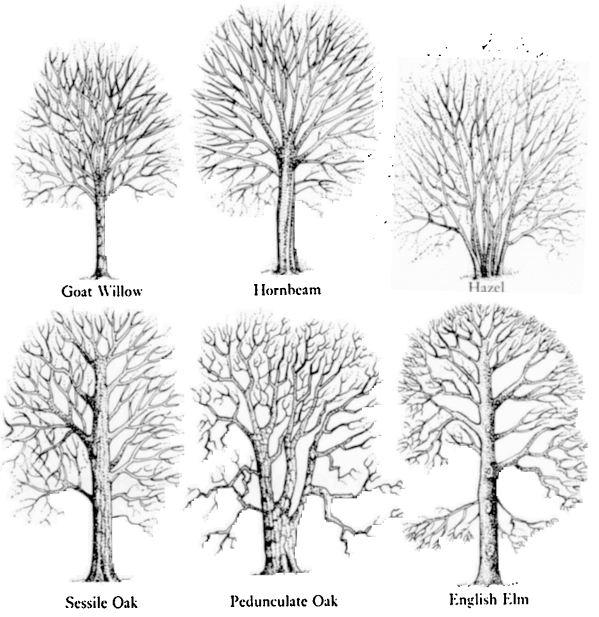
One could characterize most of the work illustrated in this book so far as totally geometric. Some of you might go so far as to claim that these designs—although visually surprising and intellectually interesting—are *excessively* geometric, that they lack "life" and have no organic qualities.

You might be ready to ask whether the shape notation capability of Logo can produce designs that share visual characteristics with pieces of the living world. Chapter 7 will argue that this question can be answered "yes."

This exercise introduces a tree-drawing problem that will be developed at length in the next chapter. I want you to think about it on your own, though, before you see my presentation; so don't read ahead into the next chapter.

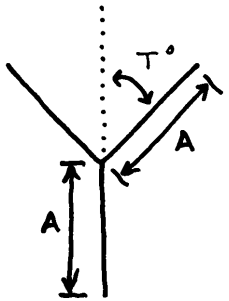
A recursive definition: a tree is a branch attached to a tree

Branching is not the only characteristic that make trees trees; there are many others, and we will eventually investigate some of them. But branching is perhaps the most noticeable feature of trees. Look at the sketches on the next page, taken from a nature guide book.

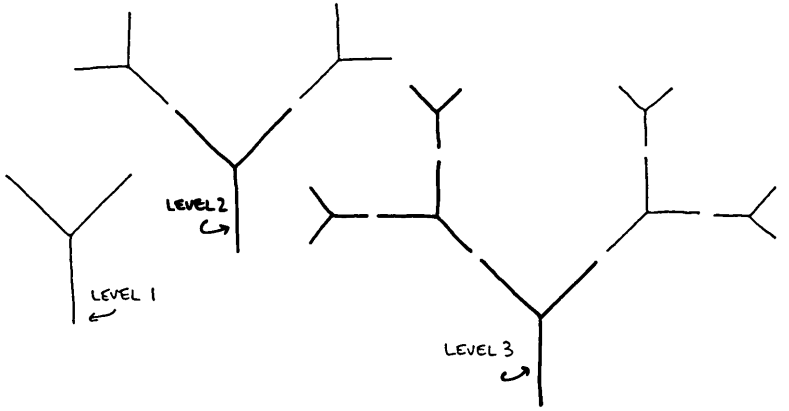


Ah, the *branchiness* of trees. So let's start with Logo procedures that draw branchy trees.

Here is a simple geometric branch. A more complex branching pattern could be tried, but let's start with the simple one first.



Let's start building a tree according to the recursive definition. Draw a branch and attach additional branches to it. Make each successive branch a bit smaller than the branch to which it is attached. We can describe these attachments in terms of recursion level:



These diagrams don't look very much like real trees yet. But remember—we are only trying to produce designs that share *some* visual characteristics with real trees; and we are starting with a single characteristic: "branchiness." Moreover, we have simplified branching into a geometric "Y" shape. We will come closer to producing realistic trees as we add more real tree characteristics. Be patient.

Your exercise is to design a recursive Logo procedure that will draw these Y-branching trees. The basic procedure, BRANCH, will draw a single branch. You may not be surprised to hear that BRANCH must be state transparent. You will be asked to explain why this is so at the end of this exercise. Be warned.

Here is the BRANCH procedure: the two arguments are as indicated in figure 1 above. Notice that state transparency is accomplished by moving the turtle back to its starting point with turtle movement commands. Using RECORD.POS and RESTORE.POS in this situation will be discussed in the next chapter.

```
TO BRANCH :A :T
 ; Draw a state transparent branch.
 FD :A RT :T
 FD :A BK :A
 LT 2*:T FD :A BK :A
 RT :T BK :A
END
```

Your work

Expand BRANCH into a recursive procedure that can draw the Y-shaped trees sketched in figure 2. Allow for an argument that scales succeeding branch size and another that controls the levels of branch attachments, that is, the level or depth of recursion.

Hint: find the places within the BRANCH procedure where you would like an additional BRANCH created.

Finally, why must state transparency be respected in this tree drawing design? Be specific.

Chapter 7
Organic Designs

"Painting is a science and should be pursued as an inquiry into the laws of nature. Why, then, may not landscape painting be considered as a branch of natural philosophy, of which pictures are but the experiments?"
John Constable

Tree experiments

I will use Constable's words to state this chapter's goal. He suggested that making pictures of our natural environment is an effective way to to see it more clearly. Although I would substitute the term *visual modeling* for Constable's *landscape painting*, I seek the same end and suggest similar means: visual experimentation with segments of our personal world can school our eyes the better to enjoy the richness of the larger world.

In this chapter I will offer you an essay on several characteristics of trees: first, their "branchiness," and second, their quality of randomness-within-a-structure. The illustrations should suggest other tree signatures to model. The exercises at the end of the chapter will ask you to justify the title of this chapter by looking at some nontree plants and beasts.

As you read through this chapter, please remember that I am showing you *my* experiments with trees; this is the way I did it. Why trees? Every Logo book has at least one recursive tree design, but few books play with the visual idea of

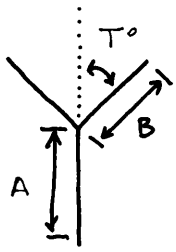
recursive trees. I hope my exercises illustrate how tiny tree machines can explore the visual implications of such an idea.

Simple recursive trees

Exercise 6.7 introduced you to the idea of drawing simple trees composed of Y-shaped branches. Let's complete that exercise so that we can get on with drawing more realistic trees and other landscape designs.

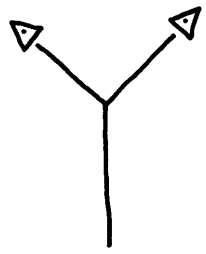
Review the BRANCH procedure introduced at the end of Chapter 6. The following procedure follows that discussion but adds an additional argument, :B. The diagram shows how :B, the branch length, relates to :A, the trunk length. Why might it be useful to have two length arguments, :A and :B, rather than just :A?

```
TO BRANCH :A :B :T
 ; To draw a state-transparent Y shape.
 FD :A RT :T
 FD :B BK :B
 LT 2*:T FD :B BK :B
 RT :T BK :A
END
```



BRANCH is reproduced again below with the symbol (***) inserted to indicate the places in the procedure where the turtle arrives at the tip of a branch.

```
TO BRANCH :A :B :T
 ; To draw a state-transparent Y shape.
 FD :A RT :T
 FD :B (***) BK :B
 LT 2*:T FD :B (***) BK :B
 RT :T BK :A
END
```



These are the two turtle positions where we would like to place another branch. This is, therefore, the position to insert some recursion apparatus. Remember that recursion only means that a procedure uses itself in its own definition. In other words, a recursive procedure asks that it be run again. You might want to refer back to the recursion diagrams of previous chapters.

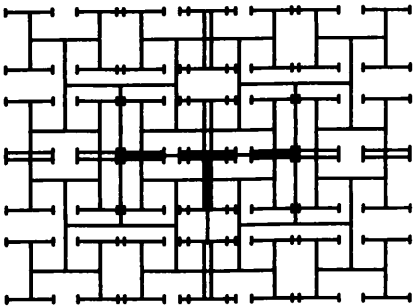
Once we have made BRANCH recursive, we will want to stop the recursion at an appropriate level. We can do this by adding an argument to control the depth of recursion. Because we may also want the branches to decrease, or increase, with the level of recursion, we will need another argument to scale the :A and :B arguments as they are passed from one level of recursion to another. To keep the names short, call the level argument :L and the scale factor argument :F.

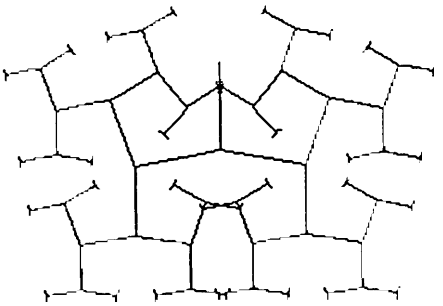
You should find the following procedure easily understandable in terms of Logo syntax. But you may not be able to guess exactly how Logo will draw the shape on the screen until you do some experimenting. Since we are making simple trees from individual branches, let's rename our procedure S.TREE. Here it is:

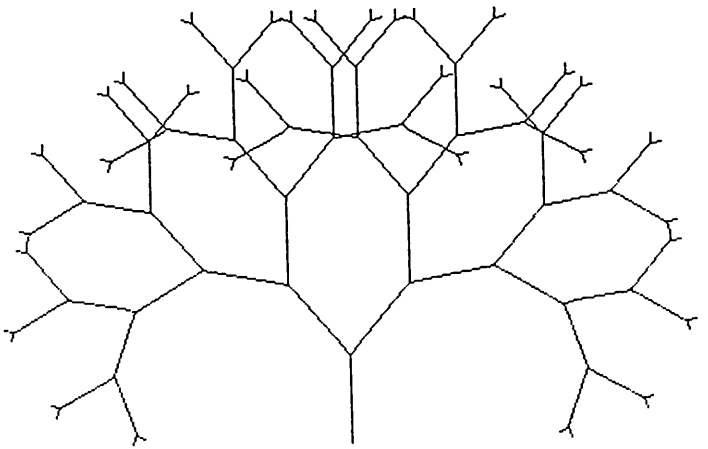
```
TO S.TREE :A :B :T :F :L
 ; To draw a simple, recursive Y-branched tree.
 IF :L < 1 [STOP]
 FD :A RT :T
 FD :B (S.TREE (:A*:F) (:B*:F) :T :F (:L-1)) BK :B
 LT 2*:T FD :B (S.TREE (:A*:F) (:B*:F) :T :F (:L-1)) BK :B
 RT :T BK :A
END
```

That's it. But to use it creatively will take some imagination and play. You might even be tempted to look at a few trees before you start your exploration. Two pages of tree experiments follow. My intent was to present a series of designs that folded and unfolded. I began with an unfolded design, a Mondrian-ish tree, and proceeded toward a more folded-up one, a cypruslike tree. Compare these designs with the folding/unfolding RECGONs of Chapter 3.

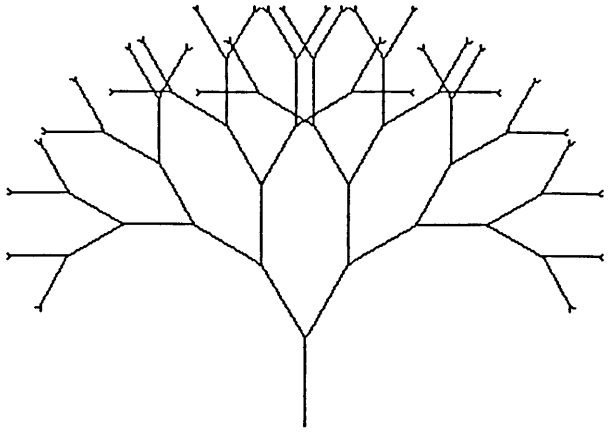
Simple tree experiments

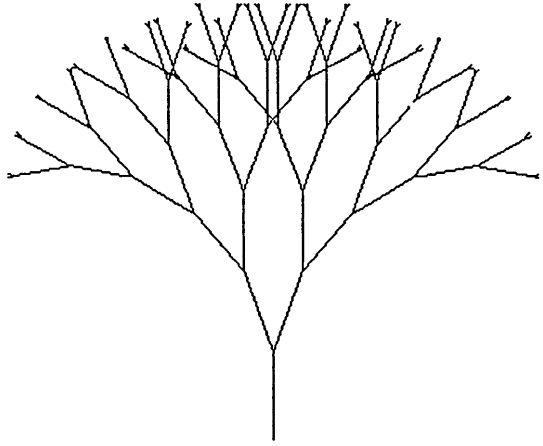


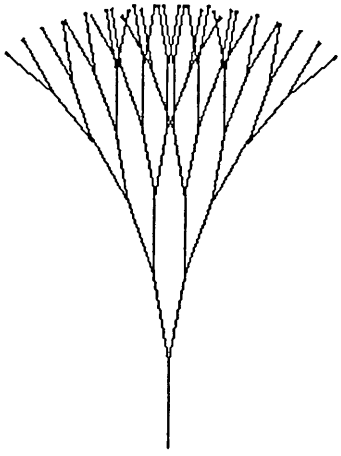




More simple tree experiments







State transparency

Let's go back to the last question of Exercise 6.7: "Why must state transparency be respected in the tree drawing procedure?" Watching the turtle as it climbs up and down the branches of your trees should supply you with an answer. If the turtle moves too fast, slow it down using the STEP command. This command lets you step through a procedure one line at a time, and you control the speed of the stepping. STEP is called TRACE in some Logo dialects. If you are not familiar with this command, review it in your Logo manual.

Another way to slow S.TREE down is to insert a time lag procedure at one or more places inside it. You might want this time lag procedure to print out the current recursion level. Would printing :L accomplish this for you?

The idea of state transparency has been useful in many of the design exercises that we have explored so far. We introduced state transparency during one of the very first exercises in this book, the centered NGON affair. State transparency was reintroduced when we began to use design recursion. Recall RECGON from Chapter 3 and the exercises dealing with the Gothic stone mason marks in Chapter 4. Later, in Chapter 5, we saw that state transparency was needed when we placed list-defined shapes within rectangular grids.

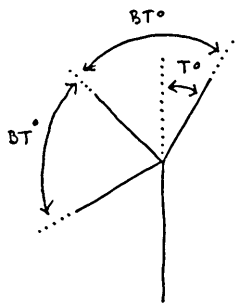
State transparency is needed here again because we are drawing trees recursively. But why does that follow? And what does the term really mean? Why *state* and why *transparency*? Here is a good place to come to grips with this concept. Can you draw a sketch of it? Would making a sketch of a concept make any sense?

Multiply branched trees

So far our trees have had a simple Y-shaped pattern; each branch forked once, producing two new branches. Why not expand this single forking design into a multiply forking one? To do so, let's add to S.TREE an additional argument that defines the number of branches leaving a fork.

Call this extended procedure M.TREE, for multiply branching tree. Its additional tree characteristic, the number of branches from a fork, will make M.TREE more general than S.TREE. M.TREE will be able to draw everything that S.TREE could draw, and more. Perhaps, too, M.TREE will create designs that look more like real trees.

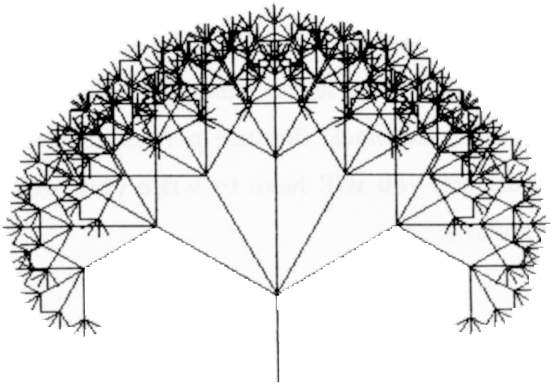
Let's call the new argument :N; it will specify the kind of forking to be used within a tree drawing. Specifically, :N defines the number of branches that leave the fork. Let's redefine the argument :T as the tilt angle that the first of these branches makes with the vertical. The new argument, :BT, will be the tilt angle between branches. The following diagram illustrates these new arguments:

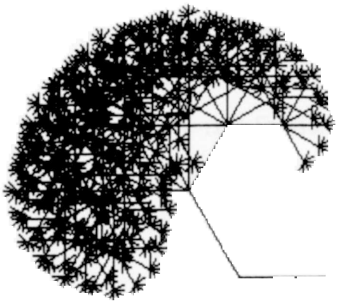


Unfortunately this new procedure will have *seven* arguments. You will need to be organized to experiment with different combinations of these arguments. You might want to design an EXPLORE-type procedure to run your experiments and place multiple trees on a single screen. (Look back at Chapters 3 and 4 if you have forgotten this powerful idea.)

```
TO M.TREE :A :B :N :T :BT :F :L
 ; To draw multiply branched recursive trees.
 IF :L < 1 [STOP]
 FD :A RT :T
 REPEAT :N [FD :B   -
            M.TREE (:A*:F) (:B*:F) :N :T :BT :F (:L-1)   -
            BK :B LT :BT]
 RT (:N*:BT)-:T
 BK :A
END
```

A few examples of multiply branched trees





Placing fruit at the branch tips

Suppose we want to put small colored shapes (fruit?) at the tips of the figures drawn by M.TREE. Lacking color, I will need a textured fruit. I'll invent two procedures, TEXTURE.ON and TEXTURE.OFF, that set the pen color to a fruity texture and then change it back to nontextured black. Because pen textures are dependent upon machine and Logo dialect, you will have to write your own texture-setters. Here's a little fruit machine:

```
TO FRUIT :SIZE
 TEXTURE.ON
 REPEAT 6 [FD :SIZE BK :SIZE RT 60]
 TEXTURE.OFF
END
```

Where should we insert this fruit machine into M.TREE? Where does the turtle reach the tips of the branches? Another way to ask this question is: Where does the turtle decide not to draw more branches? Or: When does the turtle decide that it has recursed enough? You should find this spot easily; it is located in the first line of M.TREE and is marked with the (***) symbol.

```
TO M.TREE :A :B :N :T :BT :F :L
 ; To draw multiply branched recursive trees.
 IF :L < 1    [(***) STOP]
 .
END
```

The expanded procedure is a multiply branched, fruiting tree.

```
TO MF.TREE :A :D :N :T :BT :F :L
 ; To draw multiply branched recursive trees with fruit.
 IF :L < 1 [FRUIT :D STOP]
 FD :A RT :T
 REPEAT :N [FD :A/2   -
            MF.TREE (:A*:F) :D :N :T :BT :F (:L-1)  -
            BK :A/2 LT :BT]
 RT (:N*:BT)-:T
 BK :A
END
```

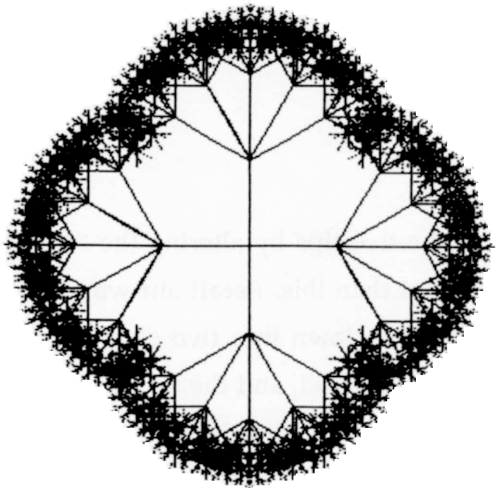
Notice a few of the changes. First, I have removed :B, the second branch size argument, and replaced it—within the procedure—with :A/2. I found that I really didn't need :B. Second, I have introduced a new argument, :D, the size of the asterisk-shaped fruit.

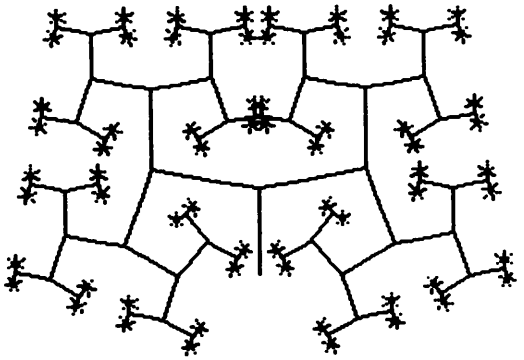
Different fruit shapes

We can change the fruit shapes placed at the tree tips by altering the FRUIT procedure. But in fact, we can be more elegant than this. Recall our work with rectangular grids in Chapter 5. We broke grids down into two elements: an underlying placement mechanism that structures a grid, and the image motif to be placed within this grid. We defined the image with a list. We could consider the current exercise to be tree grids. The M.TREE procedure is the underlying placement mechanism, and FRUIT is the image to be placed by it. This suggests the following restructuring of MF.TREE. :FRUIT is now an argument that must be a state-transparent image list.

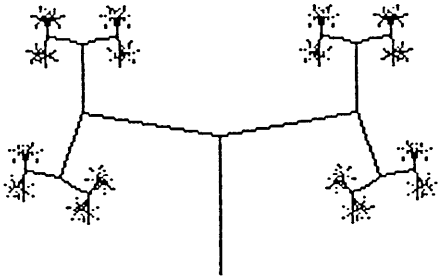
```
TO FRUIT.LIST.TREE :A :FRUIT :N :T :BT :F :L
 ; To draw multiply branched recursive trees with :FRUIT,
 ; defined by a list, placed at the branch tips.
 IF :L < 1 [RUN :FRUIT STOP]
 ; Note the change here.
 FD :A RT :T
 REPEAT :N [FD :A/2
            FRUIT.LIST.TREE (:A*:F)    -
            :FRUIT :N :T :BT :F (:L-1)  -
            BK :A/2 LT :BT]
 RT (:N*:BT)-:T
 BK :A
END
```

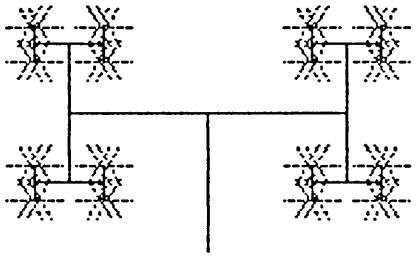
A few asterisk-fruit trees

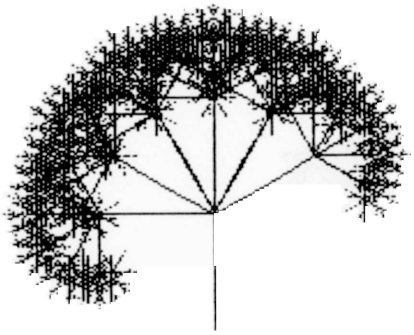


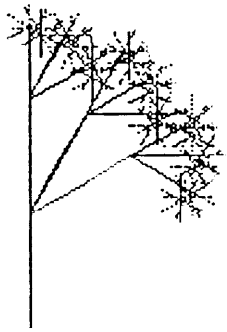


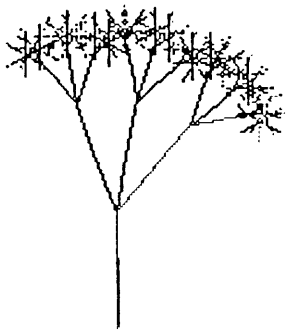
More asterisk-fruit trees



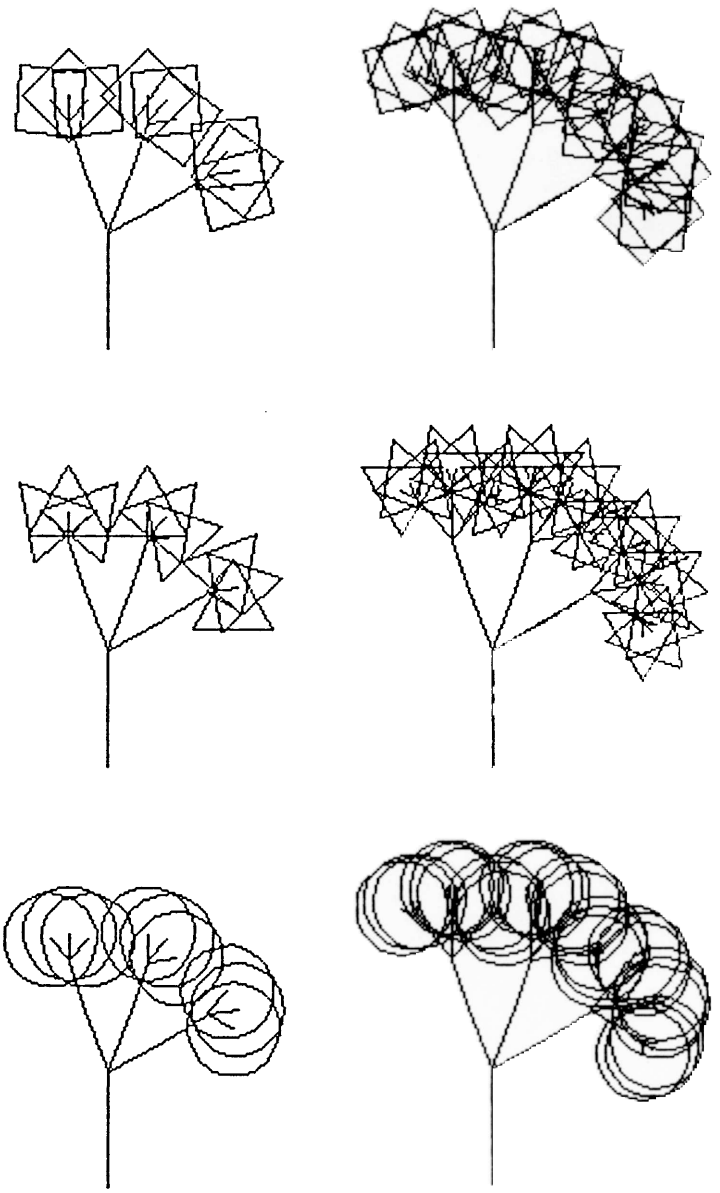






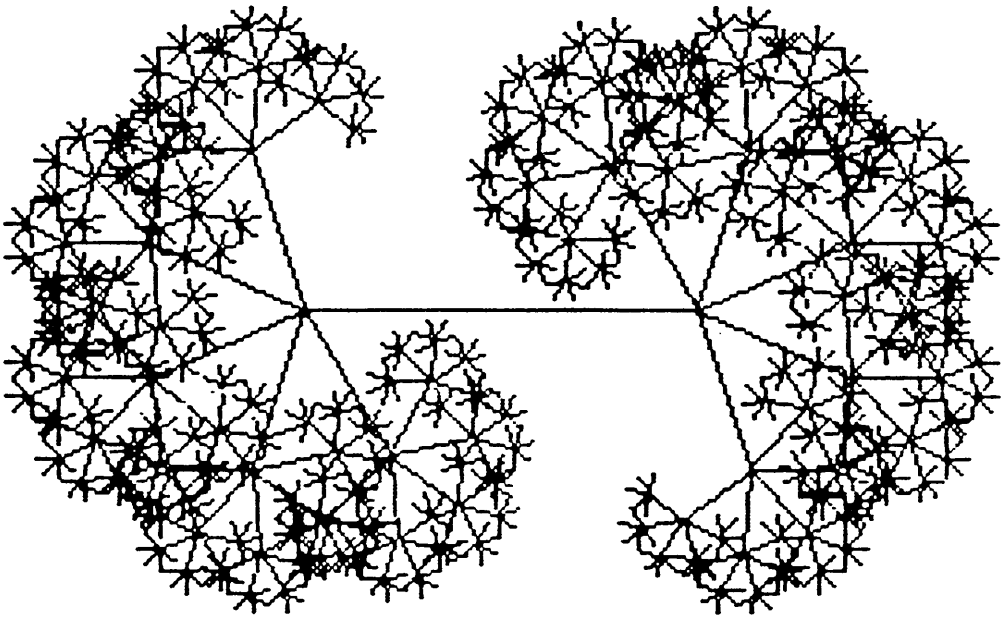


Alternative fruits

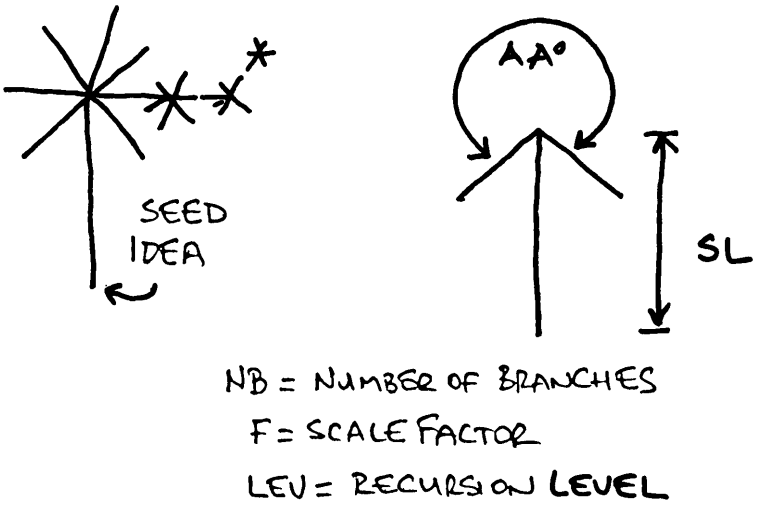


A weed model

Before going on, let's look at the procedure that produced the following illustration:



I have taken the following descriptions and sketches from the notebook of the student who did it: "I wanted to draw weeds and small plants rather than trees. I can pick a plant and take it home with me and look closely at it. Certainly easier than lugging around trees. Also, I think the lines made by Logo seem more in the scale of small weeds than gigantic oaks. I started to look at dandelions (not the flower but the flower-gone-to-seed) and a plant called Queen Anne's lace. They both exhibit a kind of recursive pattern that I have tried to model. I called my Logo procedure SEED. Here are the first few drawings that I made:



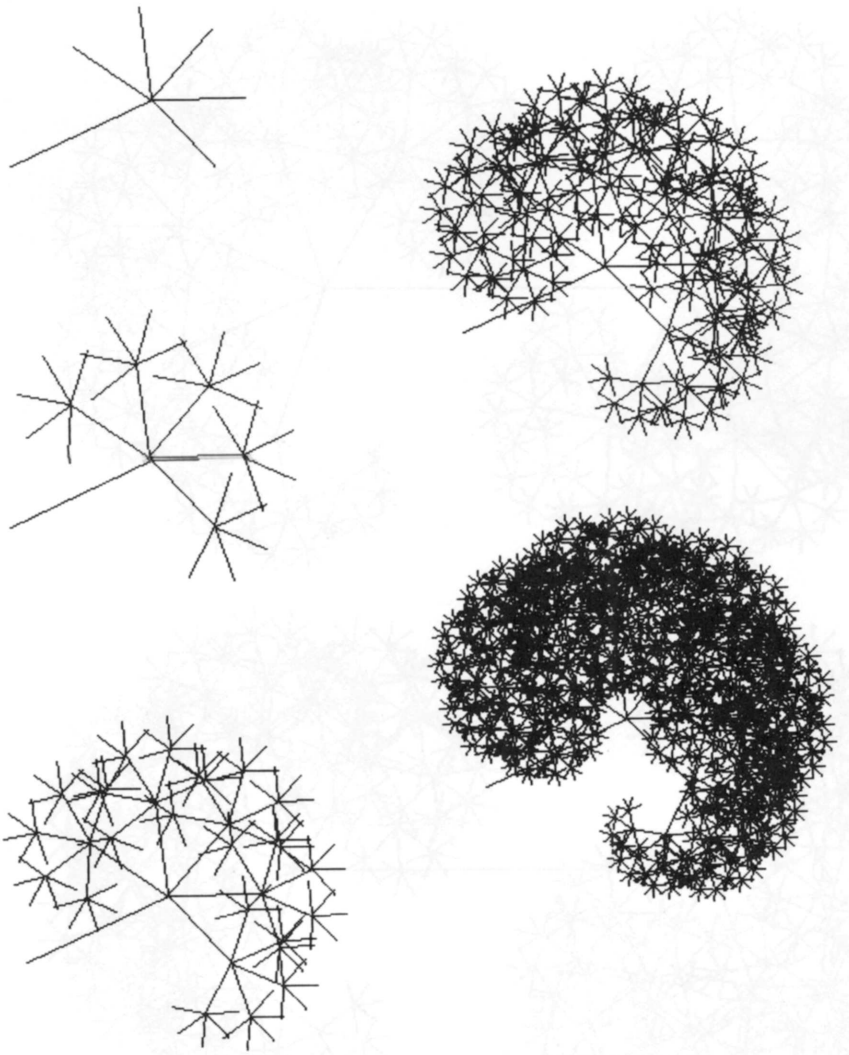
"Here is the procedure that I wrote to draw my dandelion shapes. I called it SEED because the dandelion seed shape was the image that stuck in my mind; that was the shape I wanted to explore.

```
TO SEED :SL :NB :AA :F :LEV
 ; To model dandelion flowers that have gone to seed.
 ; Try SEED 60 5 240 .6 4.
 IF :LEV < 1 [STOP]
 FD :SL
 LT :AA/2
 REPEAT :NB [SEED (:F*:SL) :NB :AA :F (:LEV-1)  -
              RT :AA/:AD]
 LT :AA/2
 BK :SL
END
```

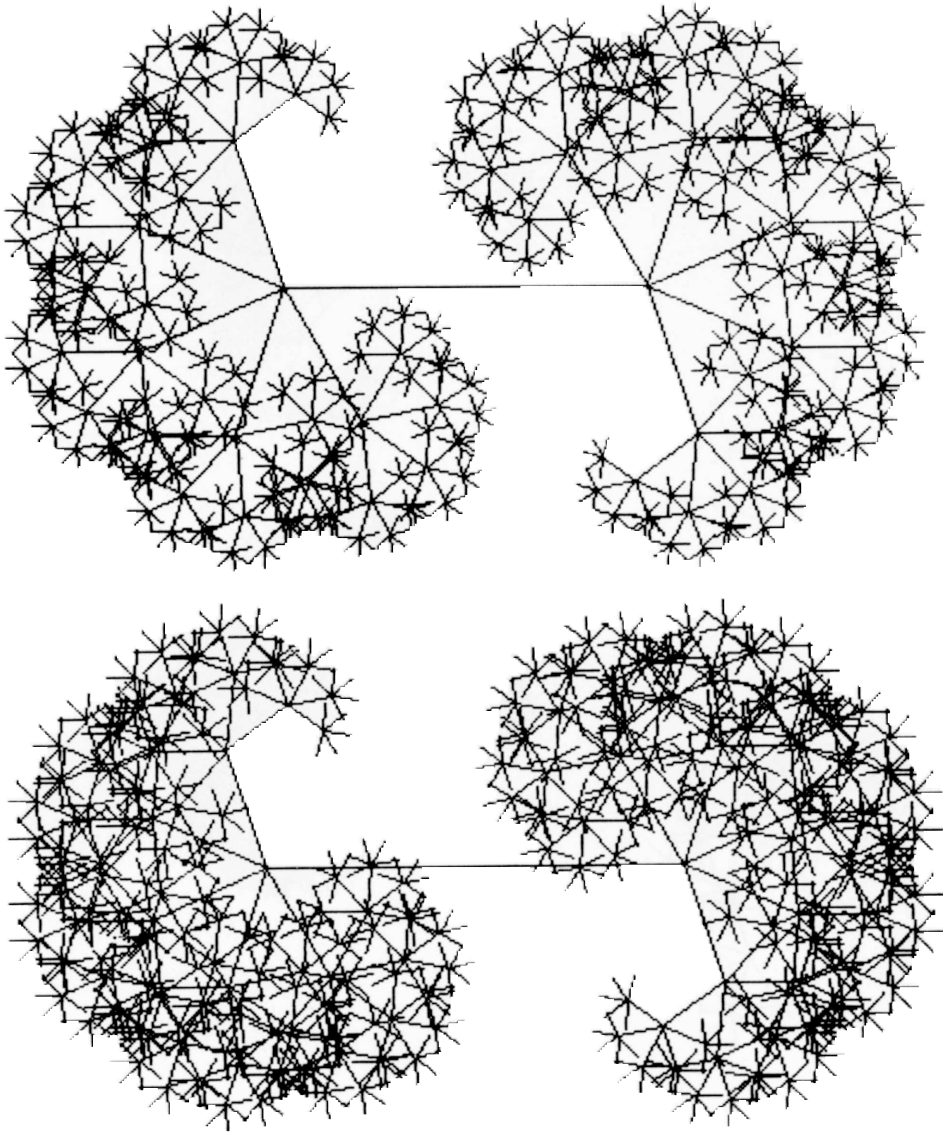
"I liked how the figures produced were not perfectly symmetric. I could have put the following turn command into the REPEAT list structure of SEED: RT :AA/(:NB-1). But I didn't like what this produced. I liked my original slightly wonky pattern.

"After running SEED with various argument values, I decided to plug two seed designs together. I wrote a little EXPLORE procedure to do this."

Seeds

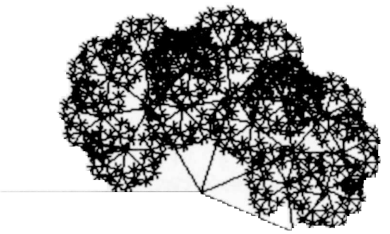


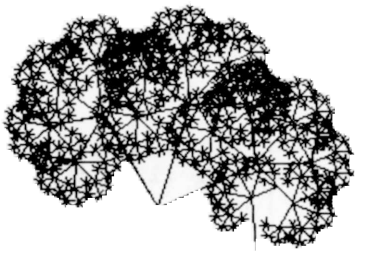
Double seeds

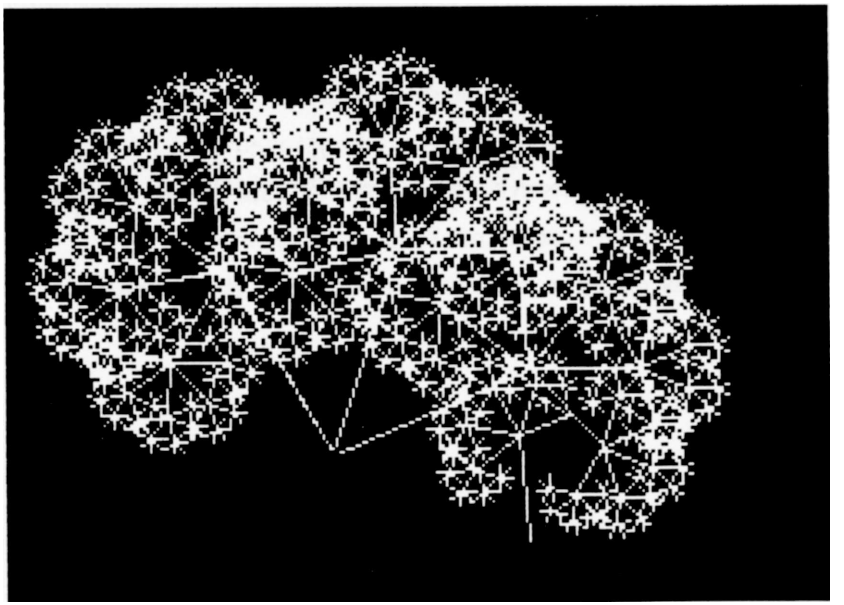


Seeds as tree clusters

"I thought I was designing seeds, but I suddenly saw the outlines of a cluster of trees in the first image below. I erased some of the lines to make it look more like several bushy trees growing closely together. I still don't like the thin trunks, though. I'll draw in some better ones with India ink."







Introducing random components into trees

"OK," I hear you say, "these trees are interesting, but they still lack a feeling of life. Real trees are not as regular as the ones presented above."

Let's introduce some randomness into these designs and see if that produces more lifelike shapes. How can we do this? We want to introduce a random component into the drawing of the :A lengths, and perhaps into the turning angle :T and the between-branch angle :BT.

We might be able to use the procedure RR that was developed in Chapter 5 to generate random grids. Recall that RR takes two arguments, :LO and :HI, and outputs a random number that falls in the range defined by these two arguments.

```
TO RR :LO :HI
 ; To output a random number in the range :LO to :HI.
 OP :LO + RANDOM (1 + :HI - :LO)
END
```

What happens if :LO and :HI are not integers? Try some funny numbers.

FD RR (.5*:A) (1.5*:A) would add some randomness into one of the tree components. The actual branch length drawn would range from .5 to 1.5 times the original value of the argument. Obviously, .5 and 1.5 are arbitrary; you can experiment with other values when we put everything together.

The problem now is how to insert RR into M.TREE. A first guess might be to put RR after every FD, BK, and RT command. Would this be correct?

We can answer this question by posing another: Is the following state transparent?

```
FD RR 5 10
BK RR 5 10
```

It certainly isn't. RR 5 10 will probably give a different value in the two situations. But we must move backward the same amount that we moved

forward. Otherwise, we don't get back to the starting point, and state transparency demands that we always get back to where we started from. We require that the same random number be used for the FD and the BK command. There are several ways of doing this:

```
MAKE "D RR 5 10
FD :D
BK :D

RECORD.POS
FD RR 5 10
RESTORE.POS
```

Let's use the first method to handle the FD :A BK :A situations within the REPEAT list structure. The RECORD.POS and RESTORE.POS can handle the getting back to the base of the branch after all the other movements have been completed. RR will be used in conjunction with the FD :A command, the RT :T command, and the LT commands within the REPEAT list structure.

But be careful. The value for :D and :POS (created inside the RESTORE.POS procedure and needed by RESTORE.POS) must be unique for each level of recursion. Why is this? We can make these two variables unique and tied to each recursion level by making :D and :POS local variables. (Review local and global variables in your Logo manual.) Local variables exist only within the lifetime of the procedure that gave them their value. Are you happy about this?

Notice, again, that I have removed the :B variable. It doesn't appear in the following procedure. As I have said previously, I found that two size arguments are not needed, and I also wanted to cut down on the number of arguments. I have replaced the :B with :A/2 as I did in the MF.TREE procedure.

Randomized trees

```
TO RAND.TREE :A :N :T :BT :F :L
 ; To draw multiply branched recursive trees
 ; with randomized components.
 IF :L < 1 [STOP]
 ( LOCAL "POS "D )
 ; For state transparency reasons.
 RECORD.POS
 FD RR (.5*:A) (1.5*:A)
 RT FD RR (.5*:T) (1.5*:T)
 REPEAT :N [MAKE "D RR (.5*:A/2) (1.5*:A/2) -
            FD :D  -
            RAND.TREE (:A*:F) :N :T :BT :F (:L-1)  -
            BK :D  -
            LT RR (.5*:BT) (1.5*:BT)]
 RESTORE.POS
END
```

Again, please notice the use of the LOCAL command in the procedure's second line. We must have a value for :POS and :D for each level of recursion. These variables will not, typically, be the same for each of the branches that make up the tree. LOCAL tells Logo to keep the variable indicated tied to each level of recursion. There will be a different :POS for each level, and there will be a different :D for each level. Logo then uses the proper value for :POS and :D according to the turtle's recursion status. This application of local variables is extremely important, but not obviously so. How do you feel about it? Can you set up an experiment to test out the logic of its use?

You might want to run RAND.TREE with line two removed (the line that makes :POS and :D local variables). What happens to the turtle and why? Make a print of the result and put it in your notebook.

Three series of random component trees

How will the random machinery introduced into the procedure `RAND.TREE` actually work? What will the designs look like? In order to see the effect of this randomizing, we must produce a series of designs, with the arguments fixed. The differences between the individual designs will be due solely to the random effects.

This exercise is similar to the random grid work we did in Chapter 5. Remember the random placement of stars? Go back and take a look.

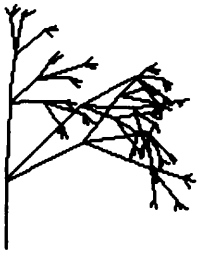
I've written an `EXPLORE`-type procedure to produce a series of designs. I have indicated the arguments used at the start of each of the series.

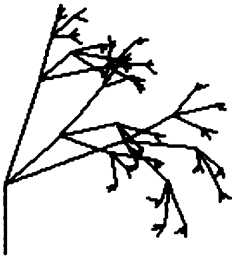
Recapitulation

What have we done *new* in this chapter? I believe we have caught some of the mystery of a living form when we mixed a little randomness into some strict geometry. What are the implications of these tree machines? Whether or not you choose to be philosophical about all this, I hope you can extend the image ideas of this chapter to simulate the shapes of some other organic shapes.

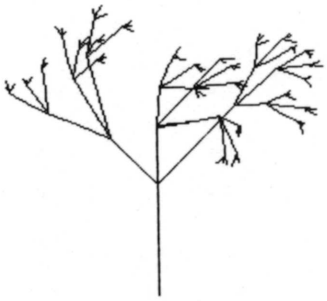
RAND.TREE series 1a

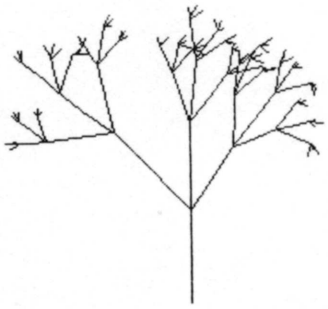
```
RAND.TREE 50 3 60 30 .6 4
```





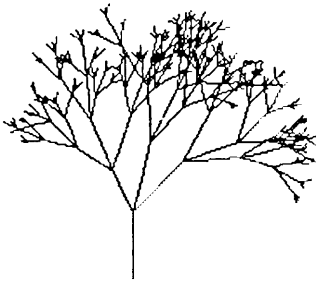
RAND.TREE series 1b

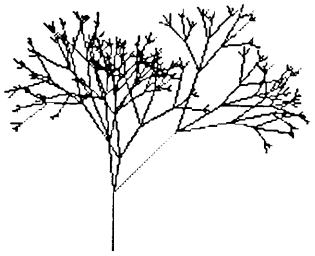


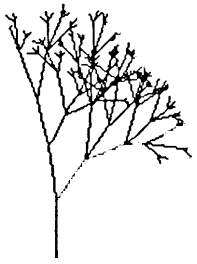


RAND.TREE series 2a

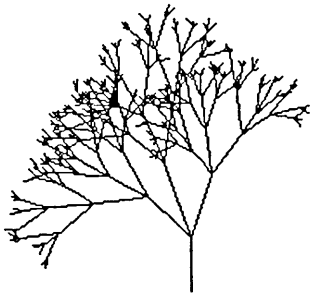
```
RAND.TREE 30 3 30 30 .75 5
```

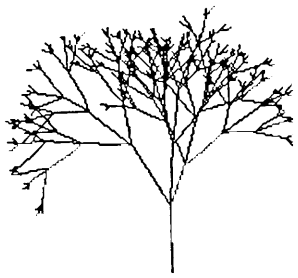


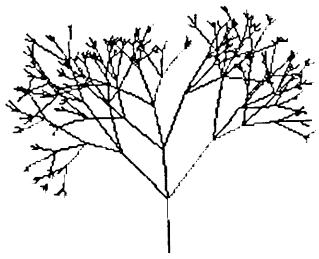




RAND.TREE series 2b

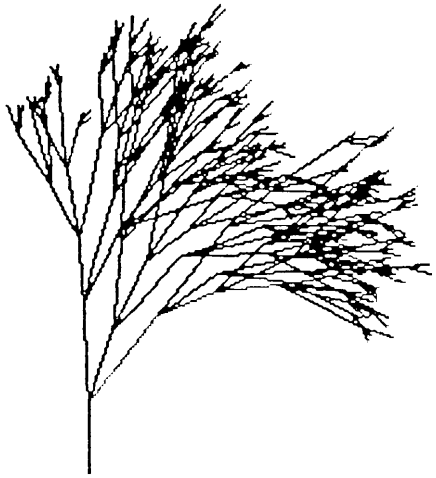


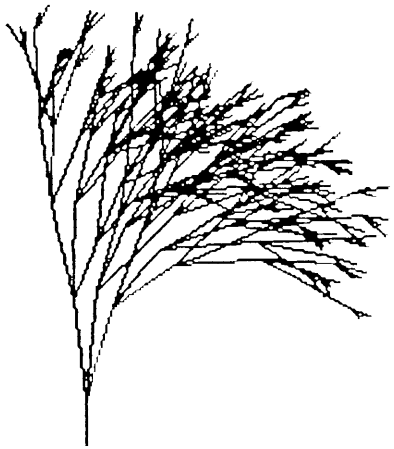




RAND.TREE series 3

```
RAND.TREE 30 3 30 15 .9 5
```



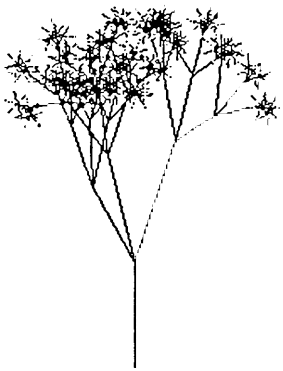


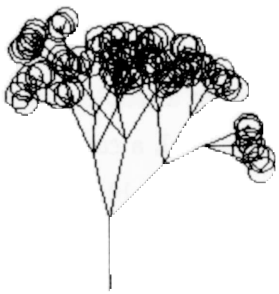
RAND.TREEs with fruit

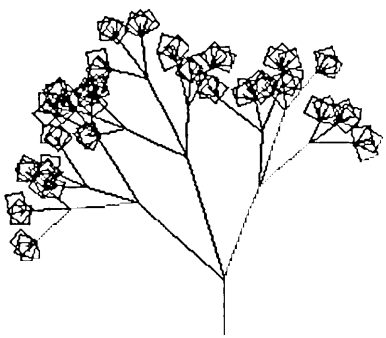
I installed a `RUN :FRUIT` component into `RAND.TREE` and added a `:FRUIT` argument. The following were produced using

```
FRUITED.RAND.TREE 60 :FRUIT 3 30 30 .6 4
```

with `:FRUIT` set to asterisks, then to circles, and then to squares.







Exercises

Exercise 7.1

Our randomized trees look more like weeds than trees because the branches are so thin. In "real" trees, the branches become thinner as they become shorter.

Design a Logo tree procedure that draws branches that exhibit thickness. Make sure that this thickness decreases as the level of recursion changes. Here are two examples of changing trunk thickness in nonrandomized trees. The first example was consciously planned by a student, and you are given two designs produced by it. The second design was "a mistake," according to its designer.

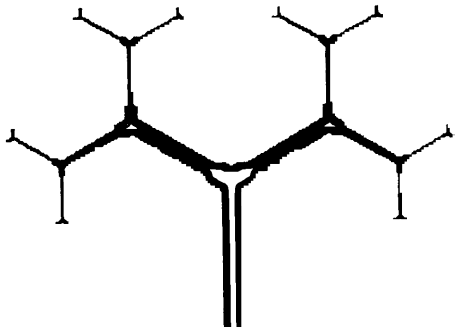
Your job is to design either randomized or nonrandomized trees that illustrate interesting variations in trunk/branch thickness. Try to do something as different from the examples as possible.

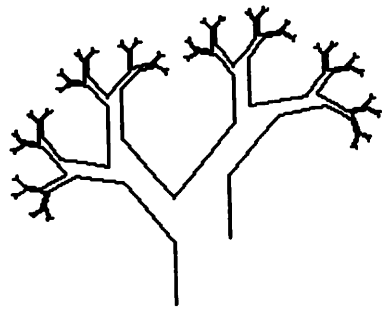
Exercise 7.2

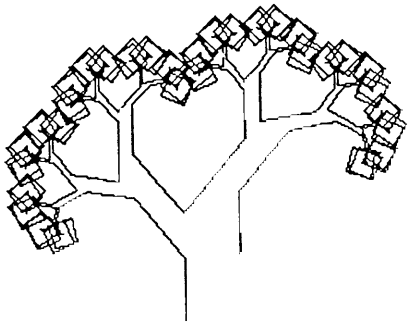
How about creating some tropical vegetation? So far, even the sticklike weeds and trees explored in this chapter are distinctively northern European. Let's try designing a "leafy something" that you might expect to find in a tropical rain forest or in a desert oasis. Here is a palm tree essay done by Paul, a computer scientist. He selected palms because they "reminded him of holidays, blue skies, blue seas, and exotic countries."

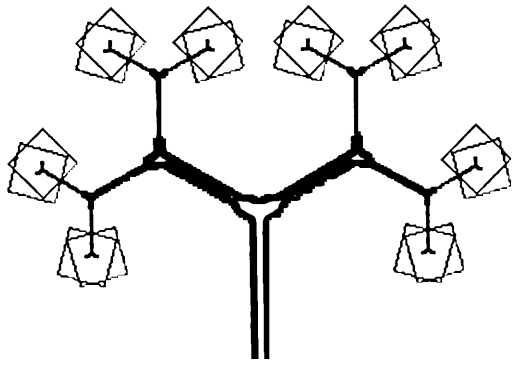
Paul characterized a palm tree in terms of the number of segments in its trunk; the rate at which these segments changed in size as the trunk lengthened (see Exercise 7.1); the length of the trunk; the "curviness" of the trunk and its tendency to bend left or right; the number of palm fronds in the tree's head, their orientation, length, and composition. The procedure structure was made recursive, and the arguments could be either fixed or random. Below are five examples of Paul's palms. You may notice that the second cluster of three trees has begun to flower!

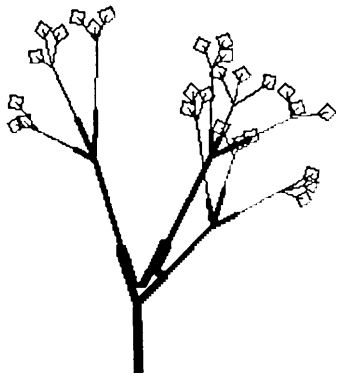
Examples for Exercise 7.1



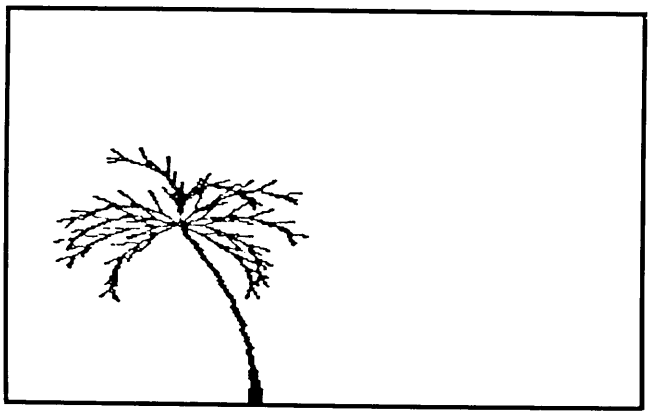


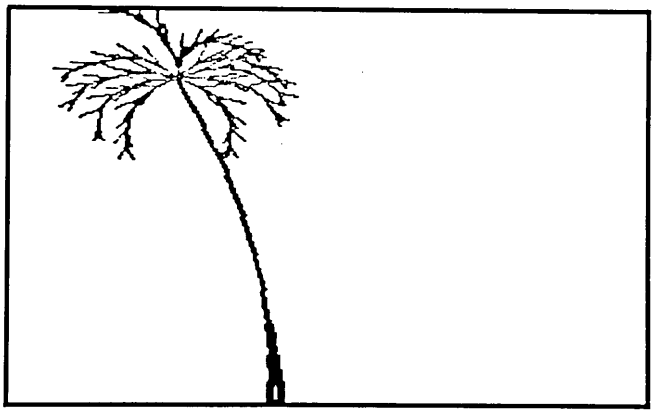


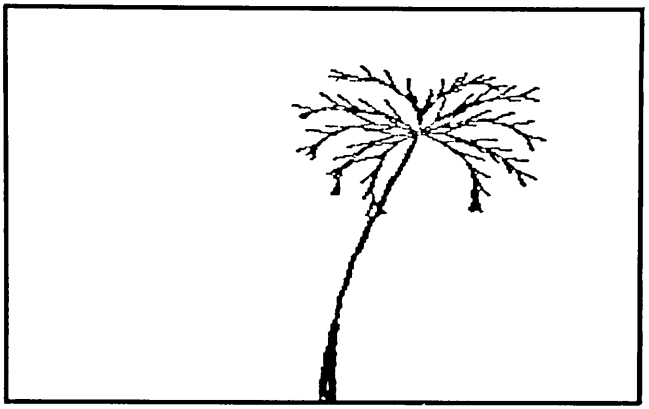


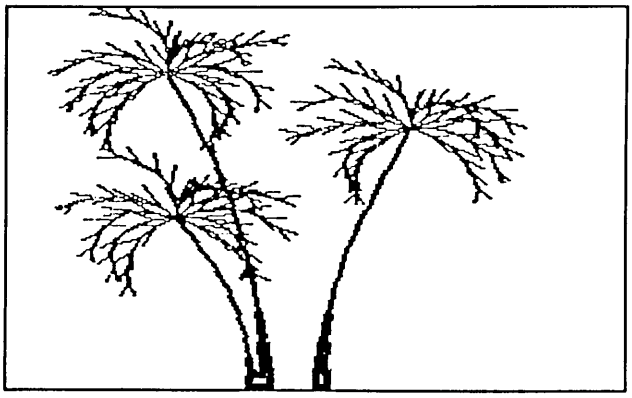


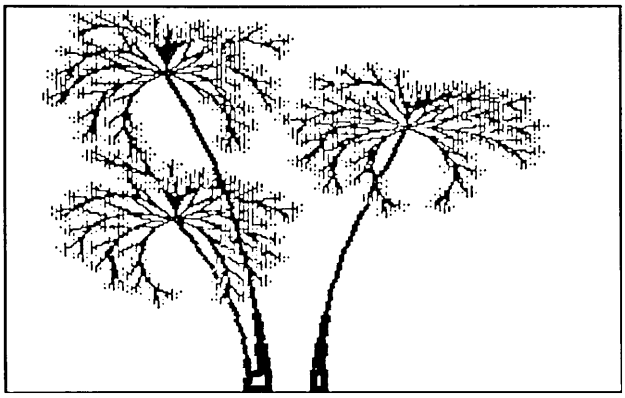
Paul's palms for Exercise 7.2



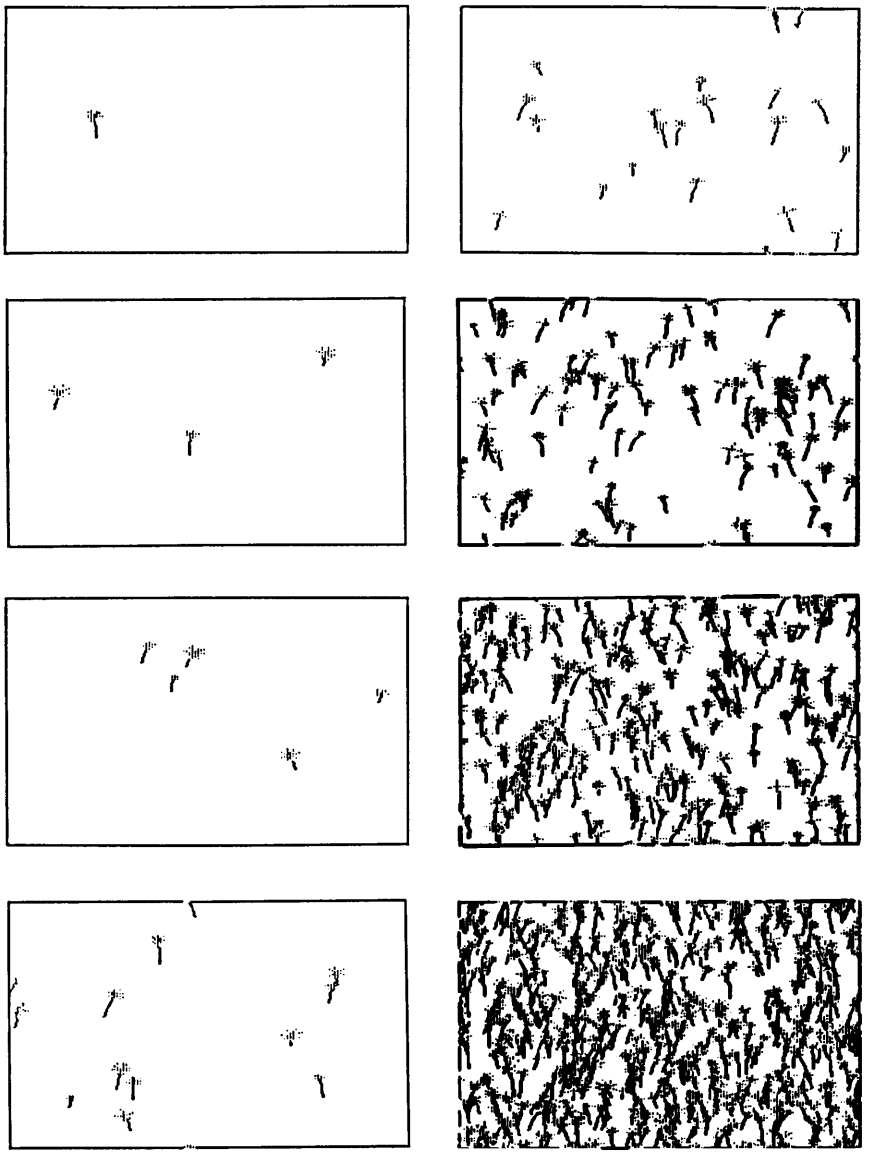








Paul then used a random placement of palm trees to simulate oases. The viewer must be in an airplane.



Exercise 7.3

Design a Logo tree-drawing procedure that prints a message on the screen to indicate the level of recursion currently being drawn. Make sure that your tree-drawing procedure has been slowed down, if necessary, so that the message can be easily read. You may want to use different pen colors to indicate the different levels of recursion.

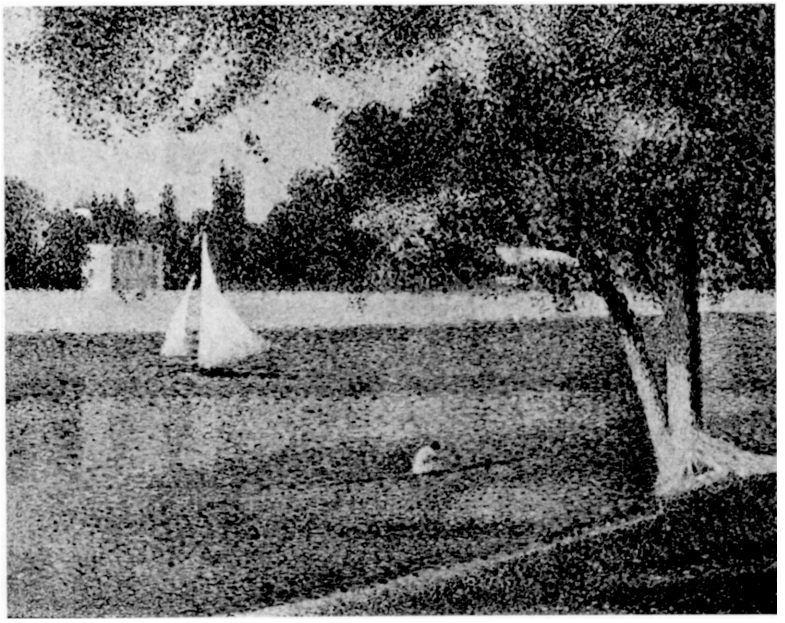
Produce a complete graphic package that visually illustrates one kind of recursion; your package could be an effective teaching tool for introducing recursion to those with no previous knowledge of it.

Exercise 7.4

What about leaves? One interesting way to try capturing the color and texture characteristics of "leafiness" is to design *pointillist trees*. What are these? The Penguin *Dictionary of Art and Artists* tells us that "According to the color theory of neo-impressionism, it is possible to obtain brighter secondary colors, such as green, by making a series of tiny blobs of both primaries (which in the case of greens would be blue and yellow) so that the blue and yellow blobs are very closely intermingled but are not actually mixed. In this way, the colors mix in the *spectator's eye* at a certain distance from the picture, giving a much brighter and clearer green than is possible by actually mixing the pigments on the palette."

At least two other sensations are felt by the "spectator" when he or she looks at a pointillist painting. First, the clouds of dots give the impression that a surface texture exists; the eyes can feel it. And second, the individual dots float in space, giving an illusion of depth. This floating occurs partially because of the natural tendency of reds and yellows to move forward and blues to recede from the viewer.

Pointillist paintings for Exercise 7.4



George Seurat (1859-1891) is the best-known pointillist painter, but other neo-impressionists of the late nineteenth century experimented with the technique. Pointillist paintings are hard to "read" in black and white reproductions; the color mixing and dot floating don't work! But you can get an "impression" of the method and the textures produced. I have reproduced here Seurat's "La Seine à la Grande-Jatte"; and second, Henry van de Velde's (1863-1957) "Faits de Village VII—La ravaudeuse."

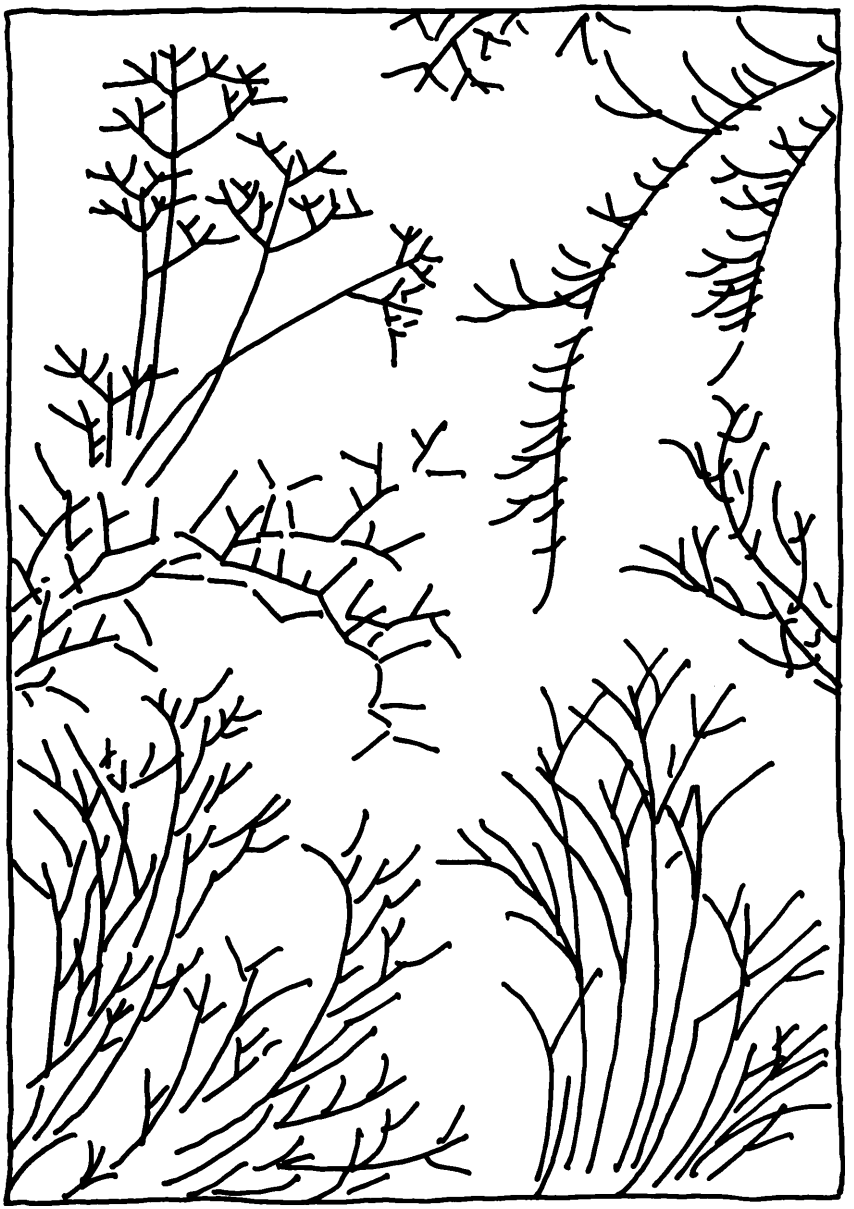
One student saw the problem of building a pointillist machine this way: "OK, let's use colored dots to represent the flash of color that might be reflected from the tree's leaves. I will draw the reflected dot-lights and not worry about drawing the individual leaves. Perhaps I can mix a number of different colored dots together in the manner of Seurat. I can place the dots randomly, too, to give a fluffy, organic shape. The randomness may give the tree a feeling of depth as well. I think I will also try to indicate the direction of the light source—from the sun; the side of the tree facing the sun should include more red dots, while the side that would be in the shade could have more blue dots."

Hint: Rework the procedures placed into the `:FRUIT` list. Make them capable of randomly placing a certain number of colored or textured dots in the "vicinity" of the current turtle position. The procedure `RR` will be useful for this. For the selection of colors or textures, you may want to give a *probability* of picking one color or texture over another. Look back at the procedure `P.GET` in Chapter 5 for some ideas on probabilistically picking things from a list. Don't forget to make the `:FRUIT` list state transparent.

Exercise 7.5

What about curved stems and branches? The following illustration is an example of Japanese pen-and-ink drawing. Does it suggest Logo procedures that use *arcs* to simulate plants? Review the arc procedures you were asked to work on in the exercise section of Chapter 5.

Japanese pen-and-ink trees for Exercise 7.5



Exercise 7.6

What about curved roots? Here are two technical line drawings of plants, both tops and bottoms! Can you characterize these plants in terms of a few arguments? How can you design and use arc drawing procedures?

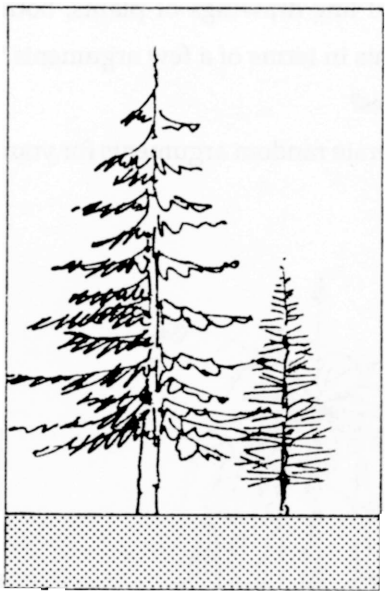
Hint: The procedure RR might be used to generate random arguments for your arc drawing commands.

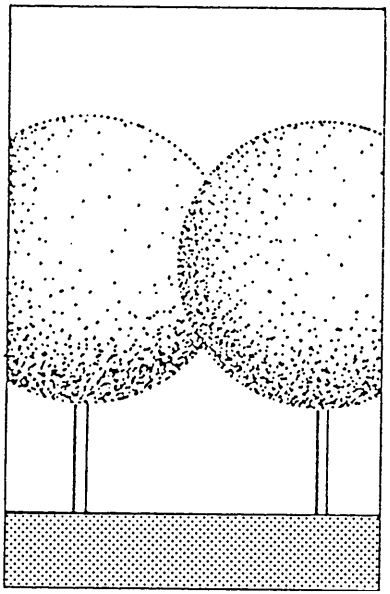
Exercise 7.7

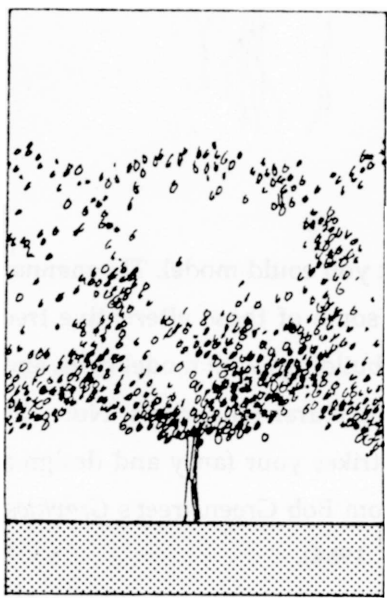
There are dozens of other tree characteristics that you could model. The manner in which architects draw trees should suggest some of these alternative tree qualities to you. Architects surround their model buildings with model trees, and the next several pages show some typical architectural renderings. Note the different approaches to treeness. Pick one that strikes your fancy and design a Logo model to do it. These designs are taken from Bob Greenstreet's *Graphics Sourcebook* (Englewood Cliffs, NJ: Prentice-Hall, 1984).

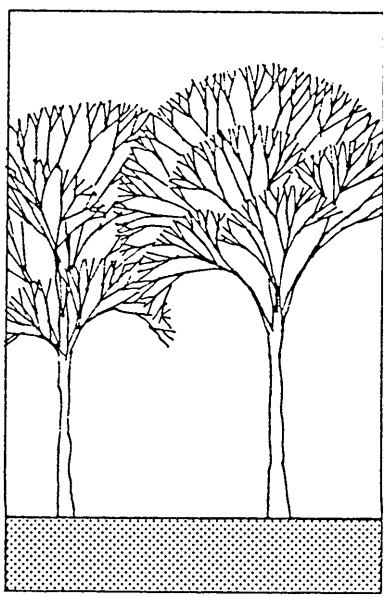
Warning: The best illustrations to use for drawing trees are real trees. Go outside *now* and look at some.

Architects' trees for Exercise 7.7









Exercise 7.8

Imagine yourself looking across a small lake. A single tree is growing on the shore just opposite you. It is night, but a full moon illuminates the tree and this image is reflected in the water. A breeze agitates the surface of the water. Design a Logo model that draws both the tree and its reflection in the lake. Include an argument for the velocity of the breeze.

Exercise 7.9

Former students have done some very interesting "Logo jobs" on fruit and vegetables. Some vegetables literally cry out for recursive techniques. You don't believe me? Go look at some artichokes, cauliflower, broccoli, asparagus! Others scream for the "oid" treatment introduced in Chapter 4 with nephroids and cardioids: go look at apples, pineapples, oranges, carrots, bananas, carrots, radishes, leeks, and turnips.

Design Logo procedures to produce a suite of designs based on the characteristics of a single favorite (or hated) fruit or vegetable. No visual hints here: go look at the real thing.

Exercise 7.10

Pick two different fruits, vegetables, or trees. Show how a single Logo procedure can draw either of the two items, as well as shapes exhibiting characteristics of both.

For example, suppose you picked a pine tree and a cauliflower. Your Logo procedures must be capable of drawing both a pine tree and a cauliflower. But your procedures must also draw figures ("caulipines"?) that are transitional between a pine tree and a cauliflower (a pine tree on its way to becoming a cauliflower). Or a cauliflower metamorphosing into a pine tree.

Pick your two objects and fill several pages of your notebook with sketches before you do any Logo programming. Do something thoroughly ridiculous, but correctly so.

Exercise 7.11

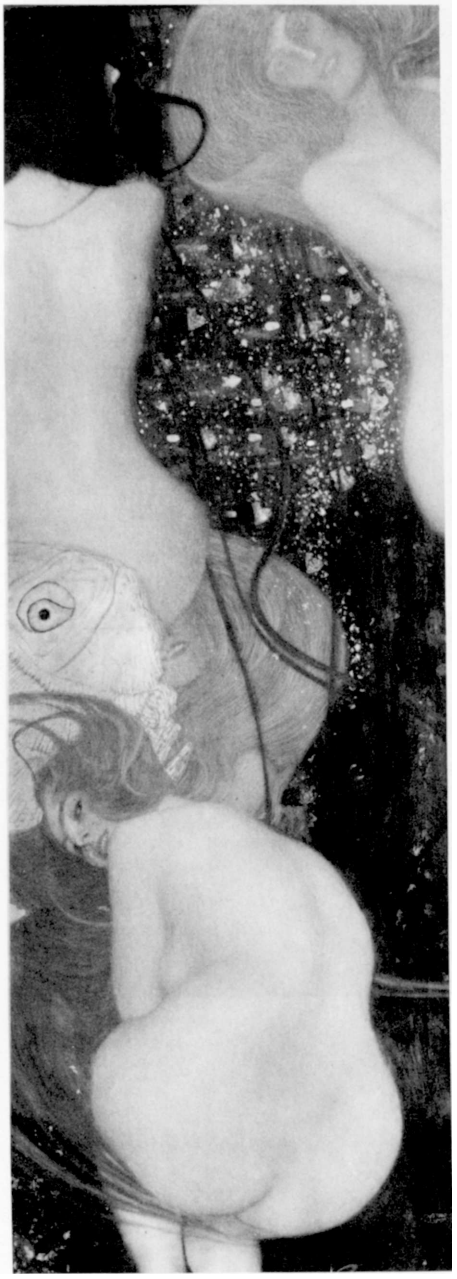
What about flowers and gardens and fields of flowers? As you may have noticed, I usually suggest two ways to begin Logo projects like this one. First, you could go out and *look at nature*. In this case that would be studying "real" flowers and gardens. Second, you could *look at a drawing or painting* of flowers and gardens. Paintings of flowers are, of course, also real and help us to see better the garden behind our house.

Let me suggest one artist who uses flower fields in almost all of his work. As always, I offer you my own taste in artists. You may see something in these reproductions that will start you off in some interesting direction. Then again, you may want to find your own artist to explore.

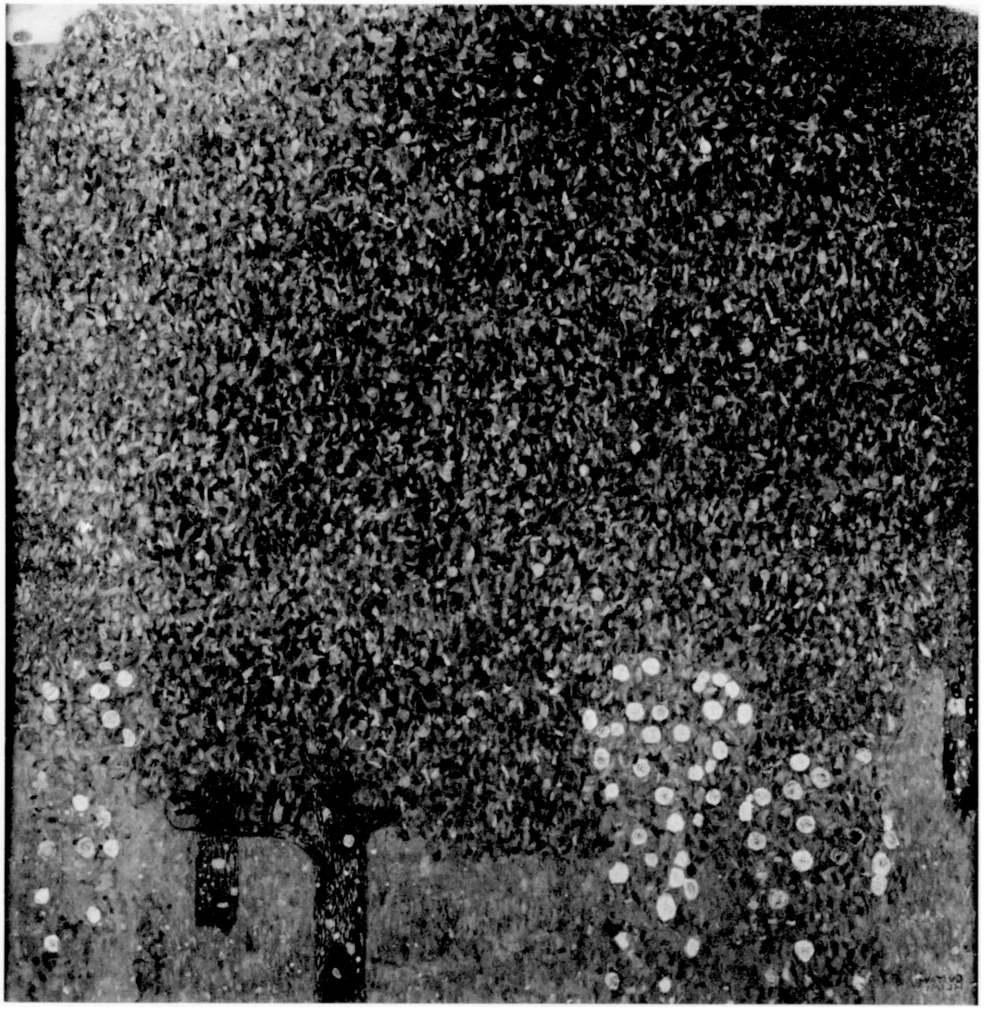
Gustav Klimt (1862-1918) was an Austrian painter who worked in turn-of-the-century Vienna. I am sure you have seen paintings like "Poissons rouges" of 1901/2, which is reproduced on the next page. Here Klimt contrasts flat geometric motifs with the molded skin textures of the very human forms. The contrast, even in this black-and-white reproduction, is striking. Each pattern type, the human flesh and the geometric forms, is made more vivid by the presence of the other.

On the page following is a painting in which Klimt uses only flower patterns; it is "Rosiers sous les arbres" of 1905. This sort of painting lacks visual excitement. There is no focus, no contrast. No surprises! Study this decorative work, though, to see how the artist has placed his flower patterns on the surface of the canvas. I would like you to Logo-produce Klimt-like flower fields, but add a surprising point of contrast. If you use naked ladies, model only a few characteristics.

A Klimt with contrasting textures



A decorative Klimt



Exercise 7.12

Try modeling one specific flower. Pick a flower that has a vivid visual personality, one that can be easily identified from a distance. Some suggestions: sunflowers, hollyhocks, iris, daffodils, tulips, daisies, lilies.

Exercise 7.13

Surround your plants with a few insects. Butterflies, moths, ladybugs, dragonflies, beetles, and fireflies might be nice.

Exercise 7.14

We've avoided animals so far. But what about some odd sea creatures? There is a snail-like creature among the imaginary PIPEGONs of Chapter 2. Find an illustrated text on sea creatures and model something like squids, jelly fish, sea urchins, or sea anemones. Something living inside shells might be a challenge, too. Keep the modeled sea creature characteristics few in number—at the start.

Exercise 7.15

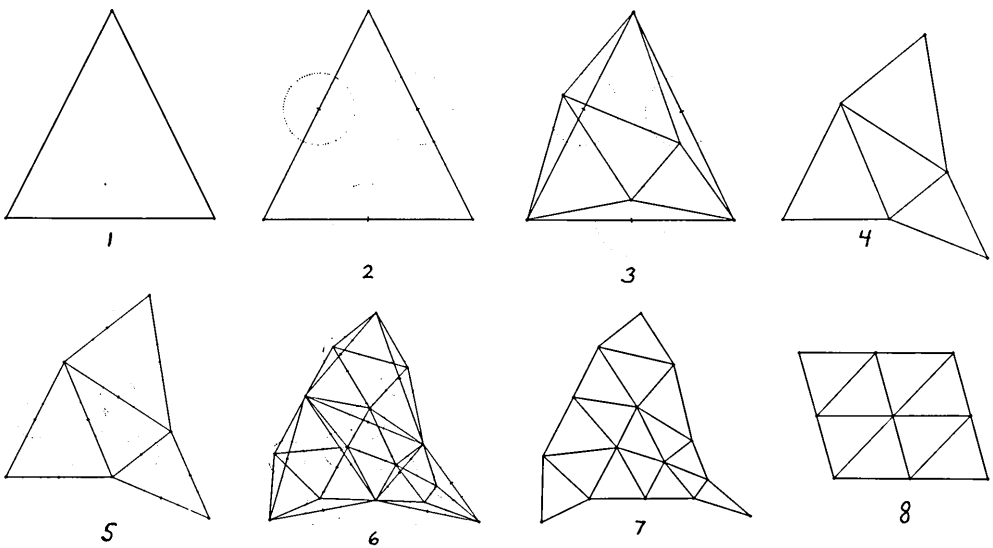
What could be more organic and more relevant to visual modeling than the human eyeball? Take a close look at your own eyes in the mirror. Make special note of the colors and patterns in your irises. Usually, on close inspection, you will find that your two eyes are different from each other and quite irregular in themselves. Design a Logo model of your own eyeballs. Generalize it to simulate a wide range of eyes that include those of your cat and tropical fish.

Exercise 7.16

Recall the FRACTALGONs of Chapter 4. We changed polygons into FRACTALGONs by changing the nature of their edges. We replaced the polygon's single, straight-line edge with an edge composed of four straight-line segments. We then replaced each of these four straight-line segment with a smaller, four-segment edge. And then each of these segments was replaced with smaller, four-segment sections. We continued to break down—or fractal—single line segments into multiple line segments until we arrived at the odd, fuzzy shapes illustrated under the heading "Some curious FRACTALGONs" in Chapter 4.

There was nothing special about replacing the straight polygonal edge with a four-segment shape; we could have used other pattern combinations of straight-line segments. But no matter what edge shape we selected, we would have followed the fractal scheme of replacing the parts of an edge with miniaturized versions of the overall edge pattern.

Let's experiment with an alternative edge pattern, and let's also introduce some randomness into fractal designing. Take a look at the following figures. Can you guess what is happening before reading the description below?



Start with the large triangle, figure 1. Imagine that a circle is drawn on the midpoint of each of the three triangle edges (figure 2). Now pick a random point within each of these three circles (figure 3), and use these points to break ("fractal") the original triangle of figure 1 into the four triangles of figure 4.

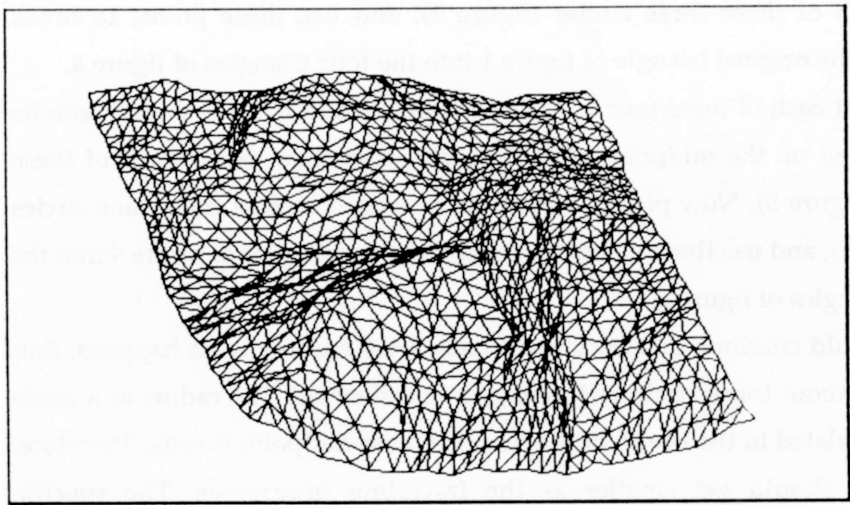
Look at each of these four triangles. Imagine that a circle is drawn with its center placed on the midpoint of each of the edges that define each of these triangles (figure 5). Now pick a random point within each of these nine circles (see figure 5), and use these points to fractal the four triangles of figure 4 into the sixteen triangles of figure 6.

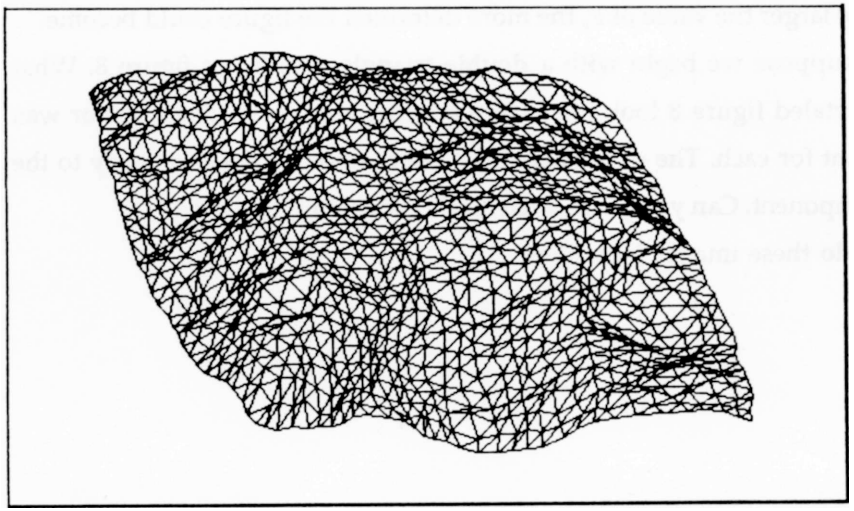
We could continue this fractaling to any level and see what happens. But, first, what about those circles? How big should they be? The radius of a circle should be related to the length of the line on whose midpoint it rests. Therefore, the circles should get smaller as the fractaling progresses. The specific relationship of the circle size to the edge length could be given by some factor, call it F. The larger the value of F, the more deformed the figure could become.

Now suppose we begin with a double triangle, similar to figure 8. What will the fractaled figure 8 look like? Here are four examples. The F factor was kept constant for each. The differences between the figures are due solely to the random component. Can you guess what level of fractaling was used?

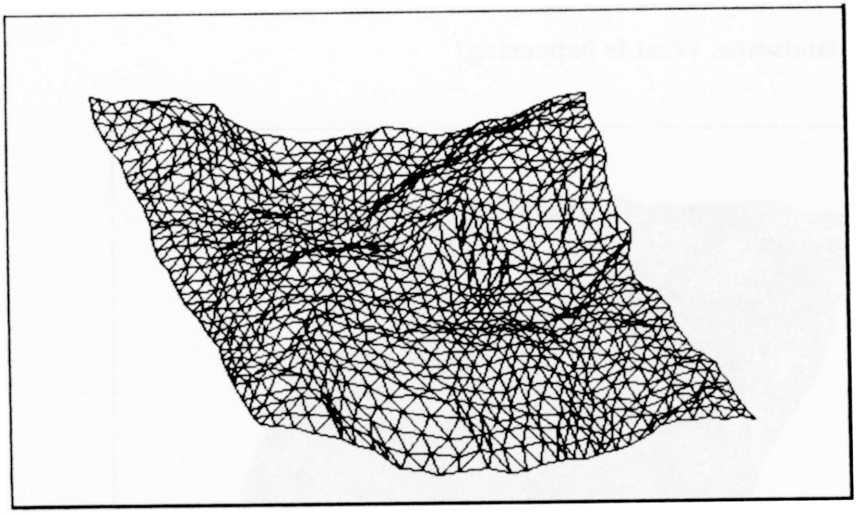
What do these images look like?

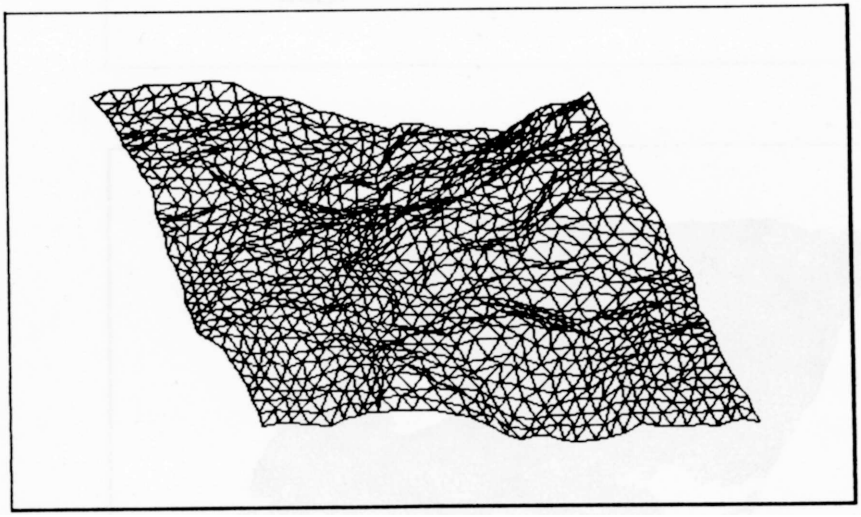
Fractal landscapes 1





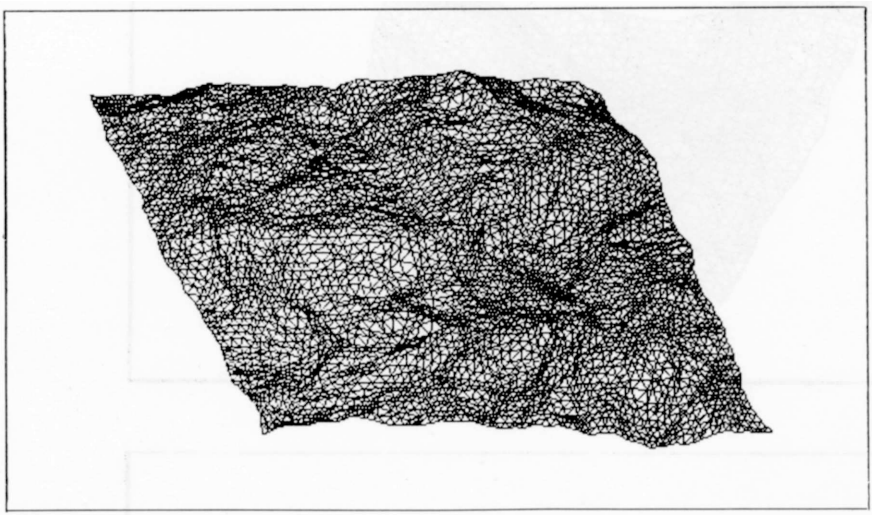
Fractal landscapes 2

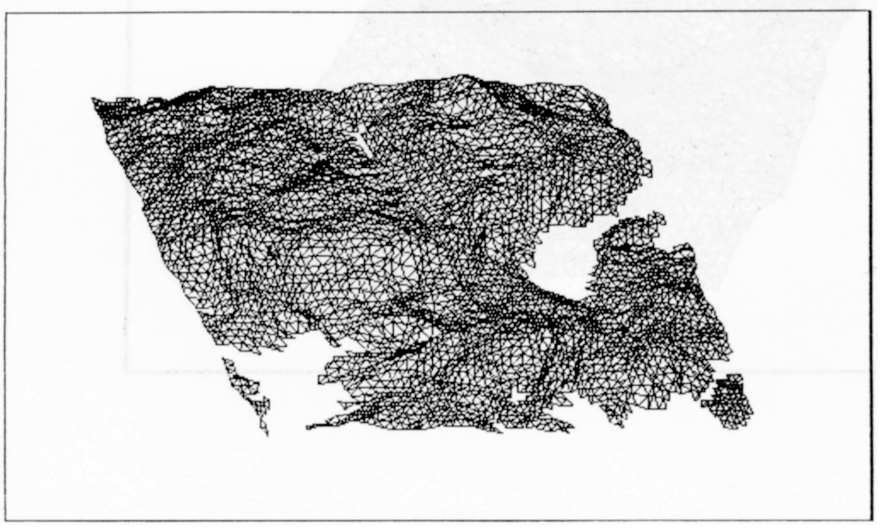




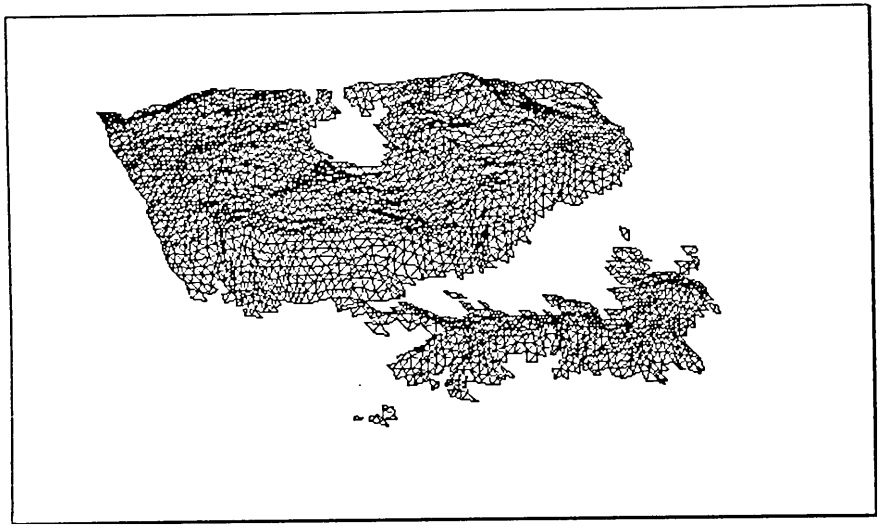
Fractal landscape before and after some rain

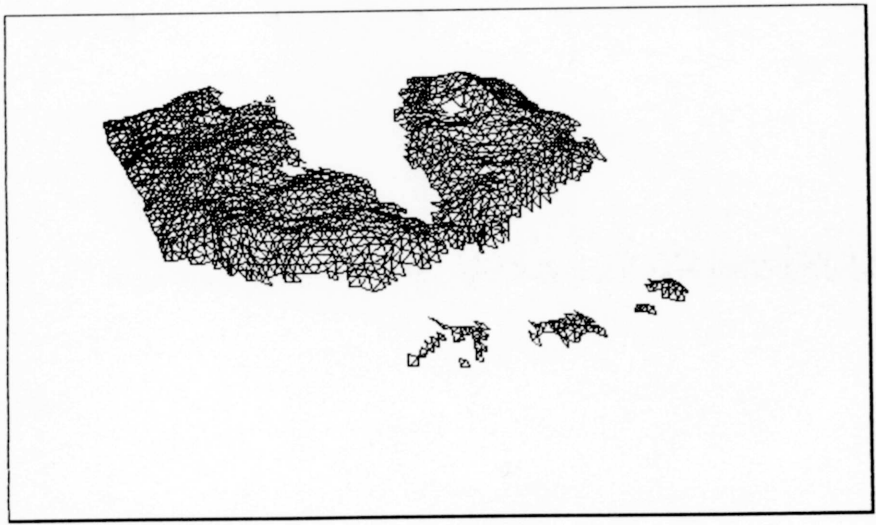
Here is one further level of fractaling. I've also decided to allow some rain to fall on the landscape. What is happening?





More rain





What's going on here? Are these scenes really mountainous? The last figure looks like a scene you might see flying into Hong Kong. Why does the eye read these figures as three-dimensional? Where *is* the eye in relation to these scenes?

How would you color in all those little triangles so that they won't look so much like chicken wire? How many triangles are there, by the way, in those figures? Recursion is obviously being used; can you give the number of triangles in terms of recursion level?

Can you put together a plan for experimenting with these visual ideas? What about a fractal mountain machine; would that be possible? What do you think the major problems would be? Put together some exploratory building plans.

Chapter 8
Space

"The question [is] whether space is real apart from space-filling objects."
W. T. Harris

Reading objects in space

Most of our work so far has been flat and two-dimensional. Our only real excursion into planning depth characteristics was the overlapping Islamic tile designs of Chapter 6. But even this was really only two-dimensional motif manipulation. However, even though we haven't been much concerned with spatial modeling, some of our line drawings occasionally seemed to *float* into three dimensions: recall some of the lacy, random tree shapes of Chapter 7 and—especially—the nephroid designs of Chapter 4.

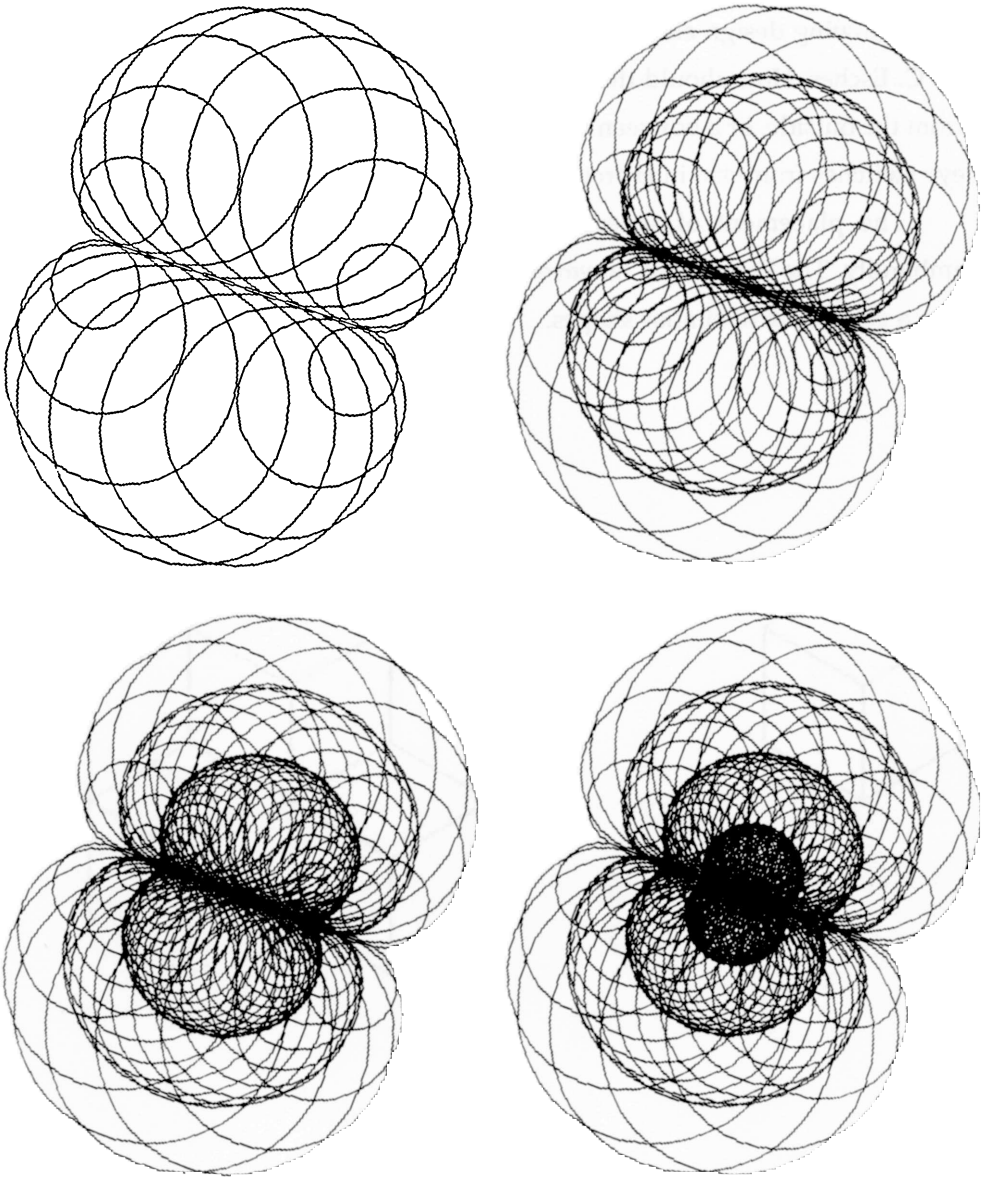
Take a look at the series of circle-based nephroids below. Even though you know that these are perfectly flat figures, do you have the impression that they are also three-dimensional and that one nephroid can actually be placed *inside* another? Why is this? Nephroids are ambiguous shapes (flat and rounded), but because the brain lives and works in a three-dimensional world, it resolves the two-dimensional ambiguity of nephroids by placing them fully into 3D space; nephroids are good examples of the brain adding space in order to read an object better. Spend some time following the lines of these odd images.

Intentional depth cues

This chapter will concentrate on some of the ways in which depth characteristics can be explicitly designed into object scenes. Art students already know how two-dimensional paper and canvas can be made to simulate space through a wide variety of methods and systems, many quite arbitrary. They know this because they have played with alternative spatial systems and conventions. If you are not an art student, you may be coming upon these ideas without such first-hand experience. So, before you are ready to incorporate depth cues into modeled scenes, you need to explore how your eyes read and interpret different forms of depth information.

We will be looking at several depth cues in this chapter: placement, shading, size variation, shape overlapping, and vanishing point perspective. The models of the chapter provide environments in which you can watch yourself reacting to messages from things in space.

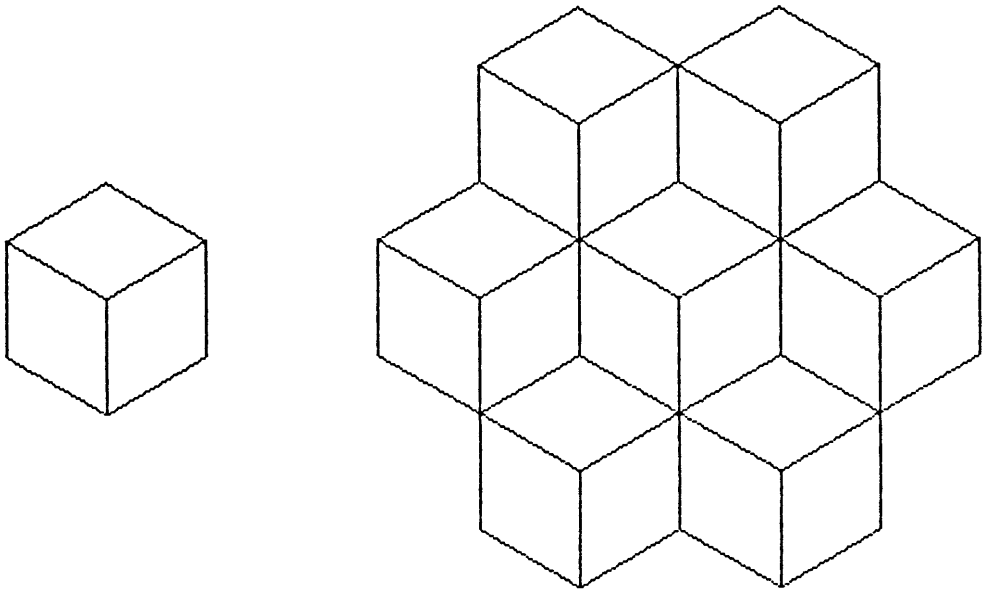
Enclosing nephroids



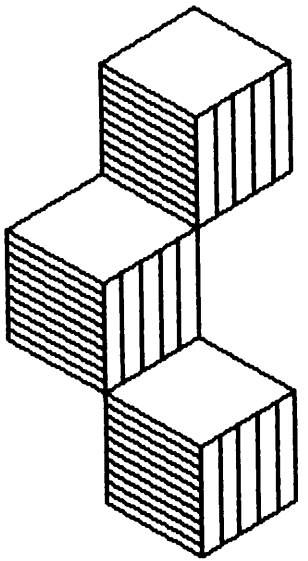
Surface shading

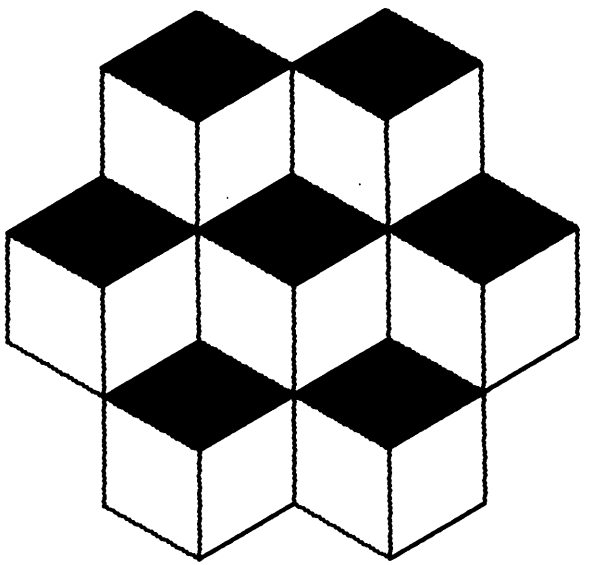
The following design element was much used by the Dutch artist and designer, M. C. Escher. How should the eye read this figure? Is it a flat design, a box seen from the outside, or a box seen from the inside? The design is ambiguous, and the eye switches from one interpretation to another.

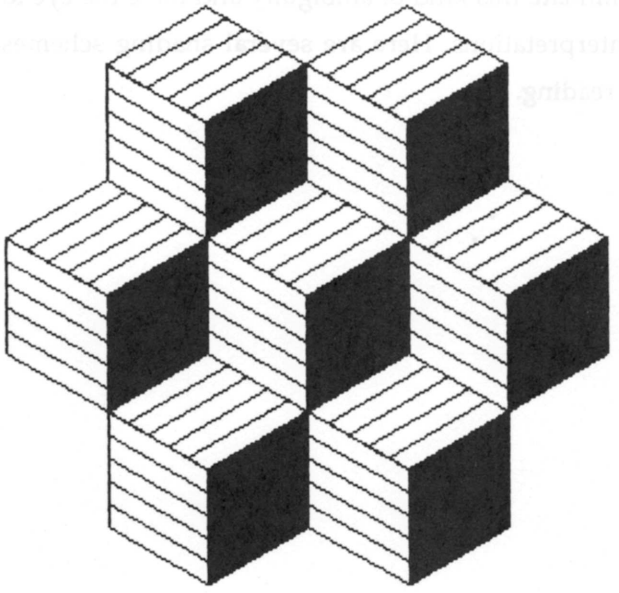
The placement of several objects into a scene can sometimes resolve ambiguity. The Escher box, however, becomes more ambiguous when it is placed into a composite of similar shapes.

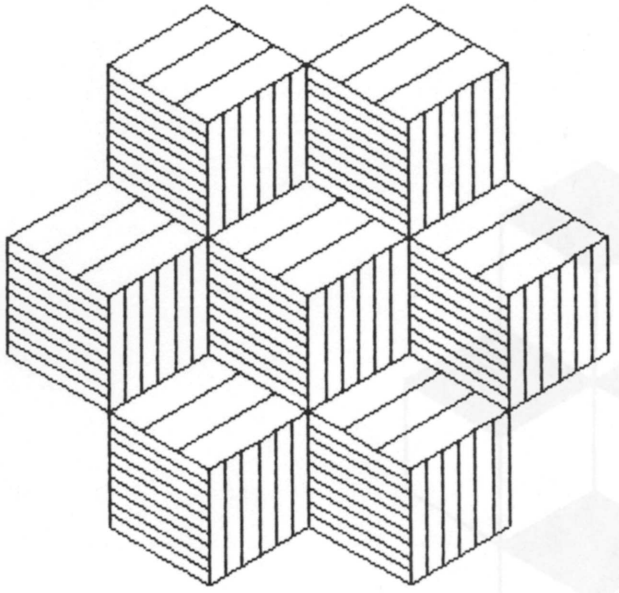


Shading is one way to eliminate this kind of ambiguity and force the eye to accept one of the possible interpretations. Here are several shading schemes that force an outside-the-box reading.







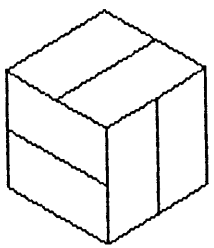


Modeling a shaded Escher box

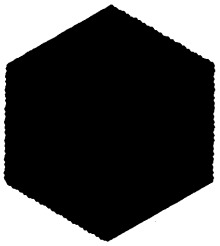
Let's write a procedure, `E.BOX`, to draw an Escher box and then shade it. We will want the procedure to be general enough to draw different-sized boxes and to alter the quality of shading on each of the three visible surfaces. The figure itself is easy enough: a hexagon in which a Y shape is placed. The procedure `E.BOX` has two arguments, `:SIZE` and `:SLIST`.

`:SIZE` is the radius of the hexagon that encloses the Y shape; it is also the length of the three arms of the Y.

`:SLIST` is a list that has three elements, one for each of the three visible faces of the box. The order of the faces is: top, right, and left. The shading is to be accomplished by drawing equally spaced parallel lines on each of the faces. Of course, the number of shading lines can be different for each face. The *number of spaces* between such lines, one number for each face, is to be placed inside the `:SLIST`. For example, an `:SLIST` of `[1 1 1]` should draw the unshaded box; each face will have only one space since an unshaded face has no lines. An `:SLIST` of `[2 2 2]` should produce a box with two stripes of space on each face:



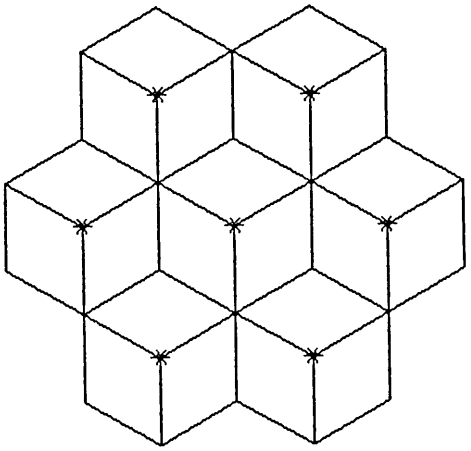
If the elements in the list are equal to the `:SIZE` argument, each of the faces should be filled completely with the current pen color—black, in my case. Here is what `E.BOX 50 [50 50 50]` should look like.



The Escher box procedures

```
TO E.BOX :SIZE :SLIST
 ; To draw a simple Escher box with parallel
 ; line shading. :SLIST must be a three-element list.
 ; Each element gives the number of between-line
 ; stripes on one face.
 LT 60 PD
 REPEAT 3 [SHADE :SIZE (FIRST :SLIST) -
           MAKE "SLIST BF :SLIST -
           RT 120]
 RT 60
 CNGON 6 :SIZE
END

TO SHADE :SIZE :STRIPES
 ; To shade a single face of an Escher box
 ; with parallel lines. The number of lines
 ; actually drawn will equal :STRIPES.
 LOCAL "D
 MAKE "D :SIZE/:STRIPES
 RECORD.POS
 REPEAT :STRIPES [FD :D RT 120 -
                  FD :SIZE BK :SIZE -
                  LT 120]
 RESTORE.POS
END
```

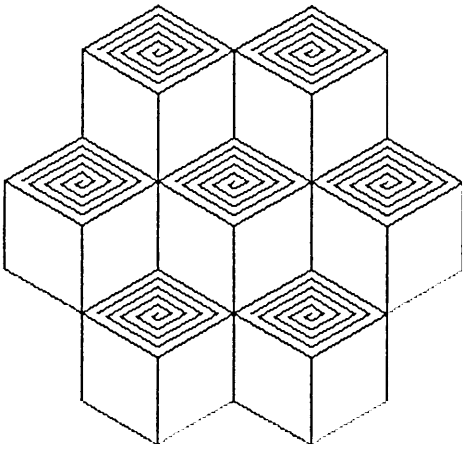
I didn't go through a turtle walk for the shading, but the method is clear. Next, let's put E.BOXs into a grid. This is similar to work we did in Chapter 6. In the following diagram the asterisks mark the centers of the E.BOXs. We need to calculate the appropriate horizontal, vertical, and indent distances between these asterisks in order to build a demonstration procedure using RIFLAG.



Because the E.BOXs are placed at the vertices of an equilateral triangle, we need to recall the relationship of the height of an equilateral triangle to its edges: height = edge*cos 30. Go back to Chapter 6 a minute and review the mechanics of NEST.DEMO, TP.DEMO, and SAW.DEMO. These three procedures were also based on the equilateral triangle. Verify the calculations used in the following:

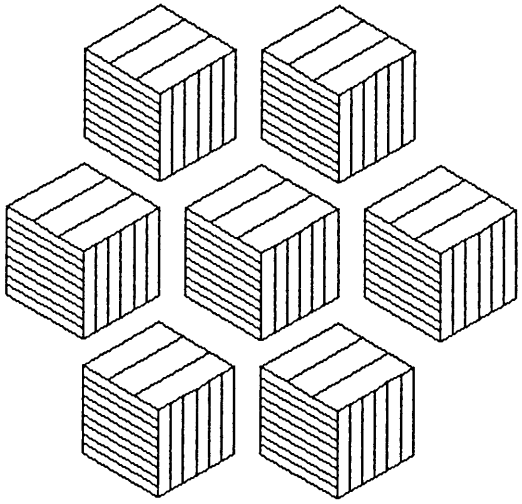
```
TO SHADED.DEMO :SIZE :COLS :SLIST
 ; To grid a series of shaded E.BOXs
 (LOCAL "EDGE "VERT)
 MAKE "EDGE 2*:SIZE*COS 30
 MAKE "VERT :EDGE*COS 30
 MAKE "MOTIF [E.BOX :SIZE :SLIST]
 GO.PT
 RIFLAG :COLS (COUNT :COLS) :EDGE :VERT(LIST (-1*:EDGE/2) 0)
END
```

One final note about my kind of surface shading. I could have used other conventions. On the next page is one example of an alternative scheme.



Placement ambiguity

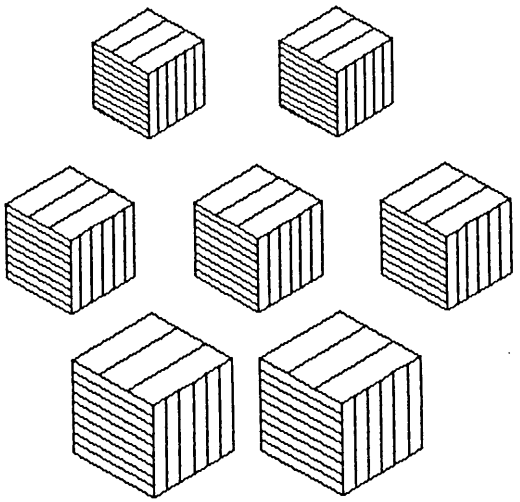
Look at the following grid of shaded Escher boxes. The grid is clearly composed of three rows of boxes, but how are the rows related in space? Is the top row floating in space above the middle row, or is it sitting on a surface and located behind the middle row? And if the three rows are sitting on a surface, is that surface flat, inclined, or stepped? This row placement is clearly ambiguous.

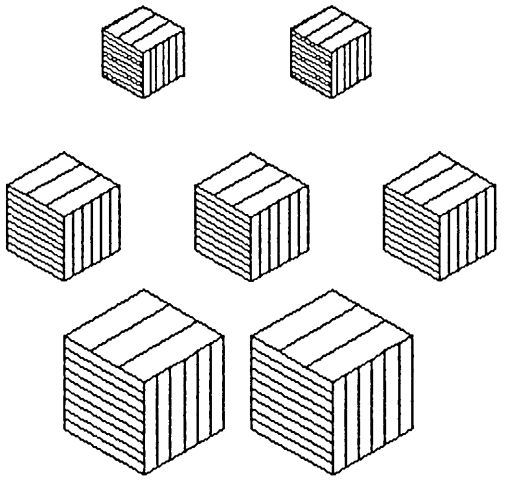


Size variation

One of the easiest ways to control the illusion of objects in space is through size variations. An object close to us is bigger than an object farther away. So differences in size can be read as differences in distance. This works fine for the space placement of objects of the same dimensions; but it is more difficult to force a single space interpretation of objects of different real dimensions, and ambiguity creeps in. For example, try to read the following two scenes. Because the shading on the boxes is the same, the eye guesses that the boxes have the same real size. Hence, the size variations are read as cues about the scene's depth, and the top row becomes the farthest away from us.

But there is an alternative reading. The top row might be small boxes that are floating above the slightly larger boxes of the middle row that are floating above the slightly larger boxes of the bottom row. Force your eyes to read the scenes in these two alternative ways. Which way seems to be the eye's preferred view? Why?





Multiple motif grids

Can you guess how the last scenes were produced? I tinkered with the generalized grid machine RIFLAG so that it could draw grids of multiple motifs. I wanted to have a different motif on each row. Then I could draw rows of different-sized (or shaded) E.BOXs to explore the illusion of receding objects . . . or whatever. I include my modified procedure to remind you about the usefulness of tinkering with the machinery you already possess, to make procedures do their jobs a little differently.

The modified machine is called RIM.FLAG for recursive, indented, multiple-motif flag. It expects a composite motif list that is composed of several list elements: there must be one motif list for each row. For example, this was the motif list used for one of the previous scenes.

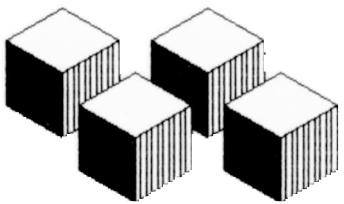
```
MAKE "MOTIF [[E.BOX .5*:SIZE [3 6 9]]  -
             [E.BOX .7*:SIZE [3 6 9]] -
             [E.BOX :SIZE [3 6 9]]]
```

Here is RIM.FLAG and its supporting procedures.

```
TO RIM.FLAG :COLS :ROWS :CDIST :RDIST :IN
 ; RIM.FLAG expects :MOTIF to have multiple
 ; motif lists: one for each row.
 IF :ROWS < 1 [STOP]
 RIM.ROWER :CDIST (FIRST :COLS)
 RIM.RSTEP :RDIST :IN
 RIM.FLAG (ROT :COLS) (:ROWS-1) :CDIST :RDIST (ROT :IN)
END

TO RIM.ROWER :C :N
 IF :N < 1 [STOP]
 RUN FIRST :MOTIF
 ; To handle the multiple motifs.
 CSTEP :C
 RIM.ROWER :C (:N-1)
END

TO RIM.RSTEP :R :IN
 PU SETX (FIRST :POINT)  +  (COUNT 1 :IN)
 RT 180
 FD :R LT 180 PD
 MAKE "MOTIF ROT :MOTIF
 ; To place the next motif into position for RIM.ROWER.
END

TO CSTEP :C
 PU RT 90
 FD :C
 LT 90 PD
END
```

Overlapping shapes

Look at the following grid of Escher boxes. There is no ambiguity about the location of the boxes. The top row is clearly behind and slightly higher than the bottom row. But how can we draw these figures?



The painter's algorithm

What we need is the so-called painter's algorithm. Why so-called? Because the method was dreamed up by computer scientists; this is how *they* think painters handle overlapping scenes. Hence the algorithm's instructions for the painter-turtle. First, determine which objects are farthest from the viewer and paint these onto the canvas. Then determine the objects just in front of the painted objects and paint those onto the canvas. Continue with this layering until you have included everything in the scene up to the viewer's nose. Use opaque paint, though, so that no object appears peeking through a layer of paint. Is this how human painters really work? Could they do it this way if they wanted to?

Whether or not this is the method real painters use, let's build a visual model using it and then draw some overlapping shapes. The first problem we encounter is how to make the turtle paint shapes that are totally opaque. Here is one way. Select the screen position where you want to draw a shape. Erase everything within the outline of the shape and then draw the shape onto this "opaque spot." The following procedures handle these tasks for E.BOXs.

```
TO BLACK.BOX :SIZE
 ; To erase everything within the outline of an E.BOX.
 LT 60
 REPEAT 3 [NIGHT.SHADE :SIZE -
           RT 120]
 RT 60
END

TO NIGHT.SHADE :SIZE
 RECORD.POS
 PENERASE
 ; Depending on your computer and Logo dialect, you may
 ; you may have to increase the pen width
 ; to erase the box space fully.
 REPEAT :SIZE [FD 1 RT 120
               FD :SIZE BK :SIZE -
               LT 120]
 RESTORE.POS
 PENPAINT
END
```

Overlapping grids

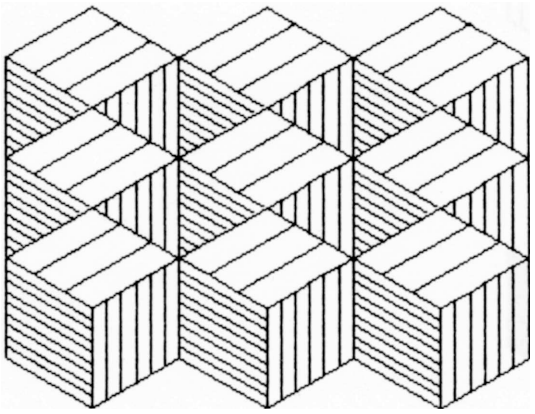
Let's organize a three-row, rectangular grid scene according to the following depth convention: the top row will be the most distant, the middle row will be in the middle distance, and the bottom row will be closest to the viewer. Our grid machine must paint the top and farthest-away row first, and then the second row of intermediate-distance objects, and lastly the row of nearest objects. If any objects in the second row overlap objects in the first row, the painter's algorithm should take charge and paint one figure over the other. We will need this help, too, with the third row, in case any third-row objects overlap any first- or second-row ones. How can we make this happen? By defining the `:MOTIF` list in the following manner:

```
MAKE :MOTIF [BLACK.BOX 50 E.BOX 50 [3 6 9]]
```

This assures us that if any image falls on top of a previously drawn figure, the overlapped segment is fully erased before the second object is painted onto the spot.

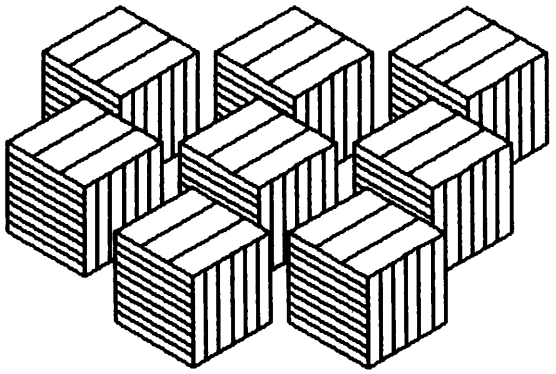
Let's try

```
RIFLAG [3 3 3] 3 (100*COS 30) 50 [0].
```

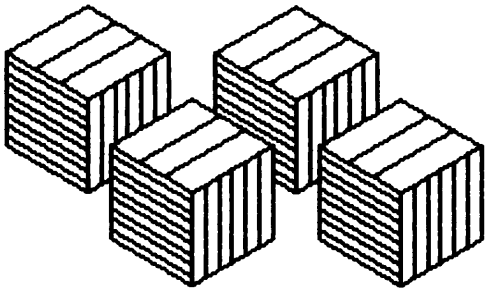


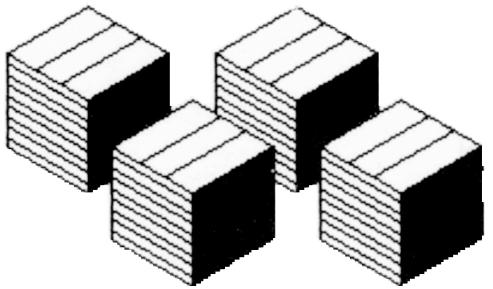
Oops. The figure is interesting but hard to read. Try moving the objects about in space a little to see if you can eliminate the ambiguities of the first image. See, for example, if it is easier to read

```
RIFLAG [3 3 2] 3 (10 + 100*COS 30) 50 [-20 40].
```

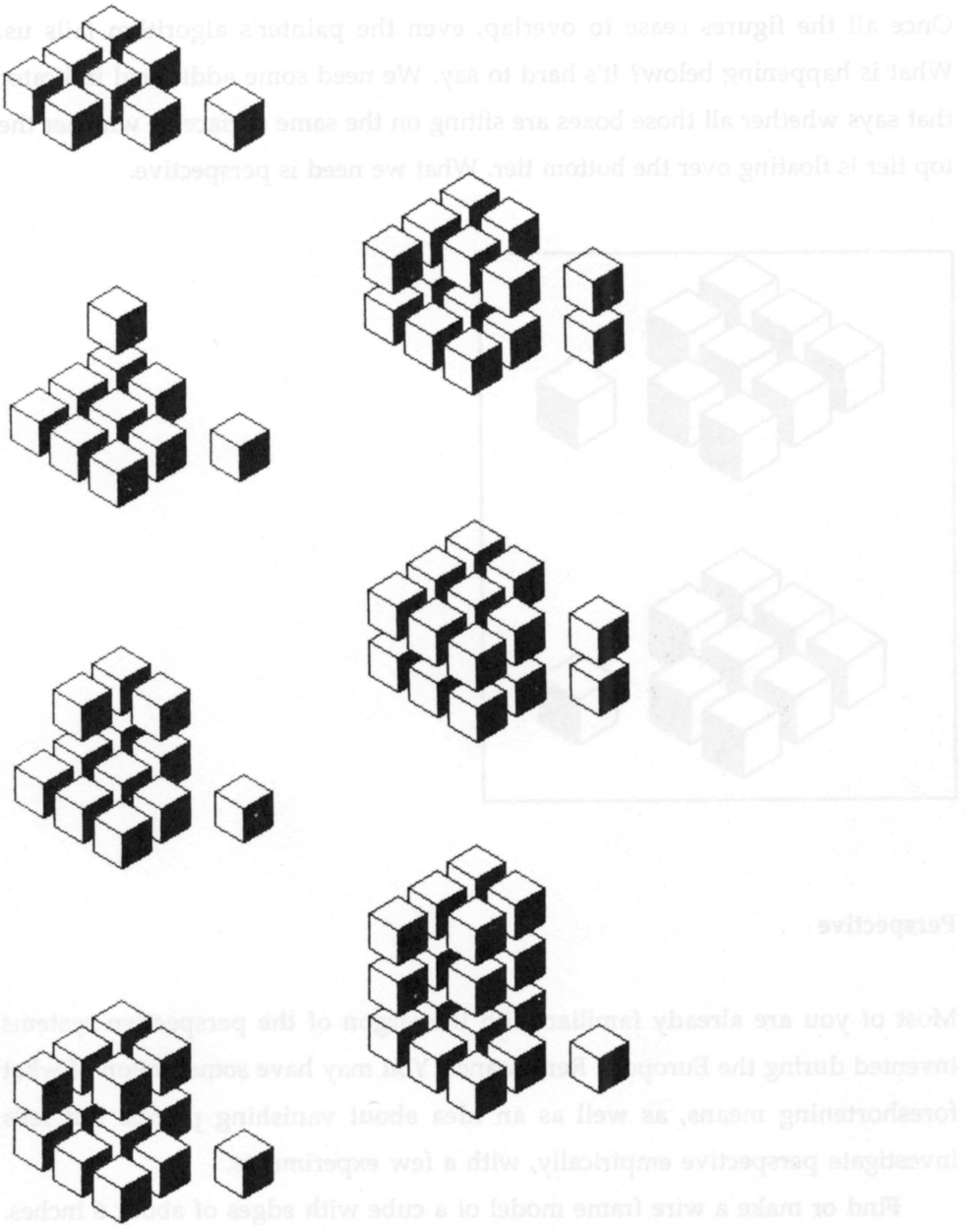


Here are two more experiments with the painter's algorithm:



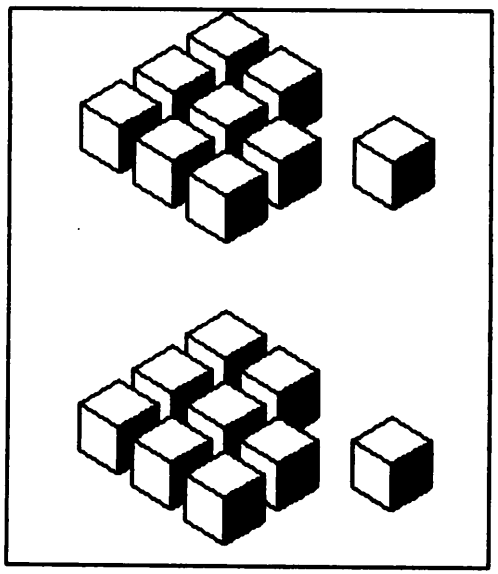


A short box play



Ambiguity again

Once all the figures cease to overlap, even the painter's algorithm fails us. What is happening below? It's hard to say. We need some additional indicator that says whether all those boxes are sitting on the same surface or whether the top tier is floating over the bottom tier. What we need is perspective.

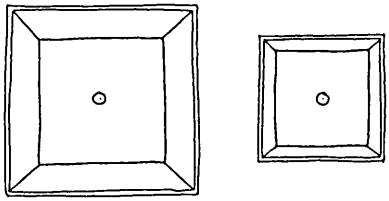


Perspective

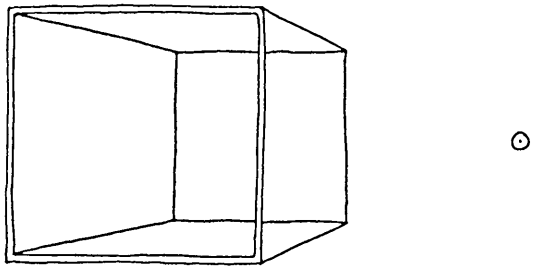
Most of you are already familiar with the jargon of the perspective systems invented during the European Renaissance. You may have some notion of what foreshortening means, as well as an idea about vanishing points. But let's investigate perspective empirically, with a few experiments.

Find or make a wire frame model of a cube with edges of about 6 inches. Hold the cube so that you are looking directly at the middle of one of its faces.

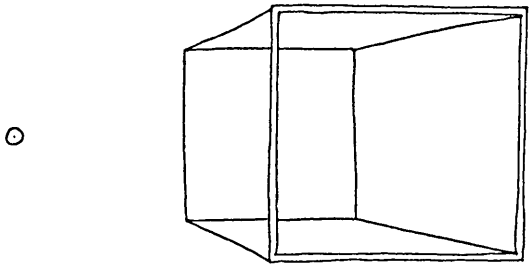
Make sure that you hold the cube so that your line-of-sight forms a right angle with the face you are looking at. Hold the cube very near your face and record in your notebook what you see. Now hold the cube as far away as possible and record what you see. Your sketches should look something like these:



Now, again look directly at the center of one face held at a right angles to your line-of-sight. This time, move the box horizontally to the left. Record your views.



Do the same as above, but with the cube moved to the right.

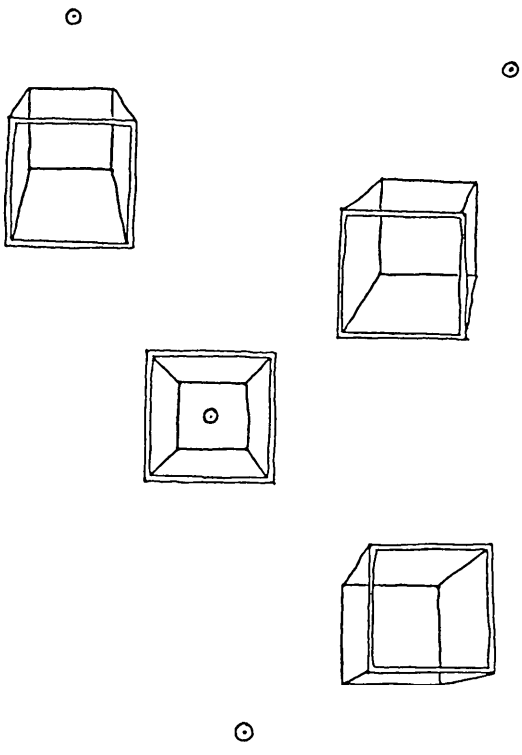


Do the same again, but lower the cube and move it to the left and then to

the right:

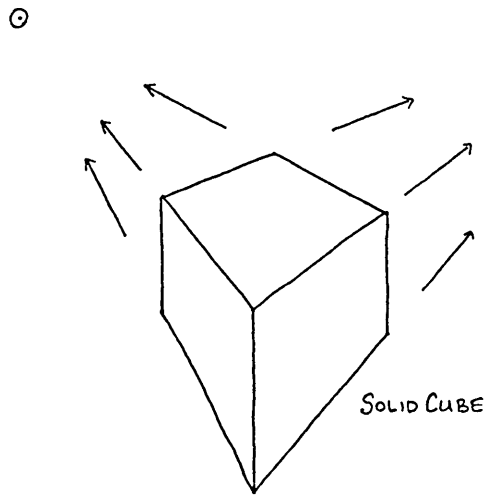
One-point perspective

You have just been witness to the operation of one-point perspective. The following sketches indicate the construction of a cube drawn with one-point perspective. Where is the viewer in each of these drawings?



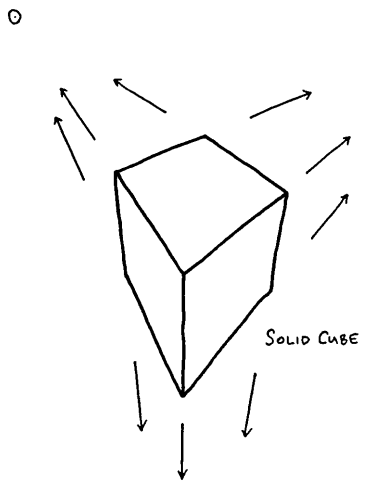
Two-point perspective

Carry out the same experiments again, but rotate the cube so that you are looking directly toward an edge rather than a face. Keep the edge vertical. The following is what happens. Where is the viewer located in this scene?



Three-point perspective

Finally, carry out the drawing experiments again, but rotate the cube so that no edges are vertical and no faces are at right angle to the line-of-sight. Yes, you've guessed it: three-point perspective. But where are you in this scene? Why are vanishing points called *vanishing* points? Can you generalize the perspective rules of the last several pages?



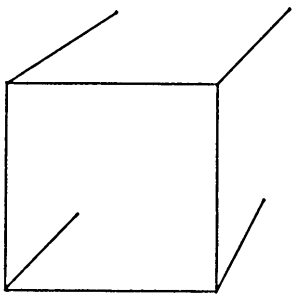
Modeled one-point perspective (almost)

How could we draw a one-point perspective version of a cube? But let's be more general than that. What about polyhedrons?

Suppose that a single vanishing point has been defined as a two-element list. Let's make it a global variable and name it :VP. The two elements will be the x and y coordinates of the point. Now let's draw a 3D CNGON. Whenever the turtle arrives at a vertex of the polygon, it should record its current position and then point toward the vanishing point. Next, send the turtle forward by an amount that corresponds to the CNGON's thickness and then return to the recorded position. Now go on to the next vertex, and so on. In Logo terms, these rules would translate into something like the following:

```
TO NHEDRON :N :RAD :THICK
 ; To draw a "kind" of  one-point perspective
 ; rendering of a 3D CNGON.
 LOCAL "EDGE MAKE "EDGE 2*:RAD*SIN (180/:N)
 PU FD :RAD
 RT 180 - (90*(:N-2)/:N) PD
 REPEAT :N [FD :EDGE RT 360/:N -
            RECORD.POS -
            SETH TOWARDS FIRST :VP LAST :VP -
            FD :THICK -
            RESTORE.POS]
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
END
```

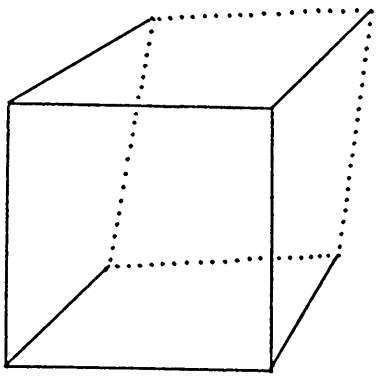
This procedure will draw figures similar to the following:



Drawing the NHEDRON's back face

Now, how do we connect the ends of those lines that have been drawn toward the vanishing point? If we knew the x-y positions of those ends, we could draw a line from the first end to the second, from the second to the third, and from the third back to the first. How? Maybe we could store these end point locations in a list. If we could put them all into a list, we could join them all together—somehow. The following procedures will let us do precisely this.

```
TO EMPTY.BAG
 ; To create a global variable named :POINT.BAG and make
 ; sure that it is empty.
 MAKE "POINT.BAG [ ]
END

TO ADD.POINT
 ; To put the turtle's current x-y location
 ;into the :POINT.BAG.
 MAKE "POINT.BAG (SE :POINT.BAG XCOR YCOR)
END

TO CONNECT :LIST
 ; To connect the x-y point stored in a list.
 IF EMPTY? :LIST [STOP]
 SETXY (FIRST :LIST) (LAST :LIST)
 PD
 CONNECT (BF BF :LIST)
END
```



Completed NHEDRON

Let's put the new list apparatus (indicated by <---) into the body of NHEDRON:

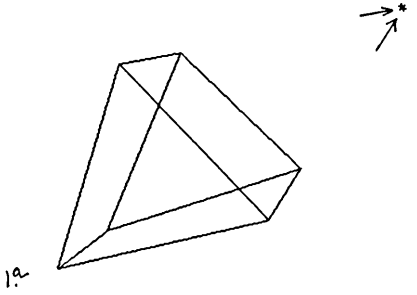
```
TO NHEDRON :N :RAD :THICK
 ; To draw a "kind" of  one-point perspective
 ; rendering of a 3D CNGON.
 LOCAL "EDGE MAKE "EDGE 2*:RAD*SIN (180/:N)
 LOCAL "POINT.BAG <---
 EMPTY.BAG <---
 PU FD :RAD
 RT 180 - (90*(:N-2)/:N) PD
 REPEAT :N [FD :EDGE RT 360/:N -
            RECORD.POS -
            SETH TOWARDS FIRST :VP LAST :VP -
            FD :THICK -
            ADD.POINT - <---
            RESTORE.POS]
 LT 180 - (90*(:N-2)/:N)
 PU BK :RAD PD
 RECORD.POS <---
 PU CONNECT :POINT.BAG <---
 SETXY (FIRST :POINT.BAG) (FIRST BF :POINT.BAG) <---
 ; Connect last end point to first end point. <---
 RESTORE.POS <---
END
```

One-point perspective experiments

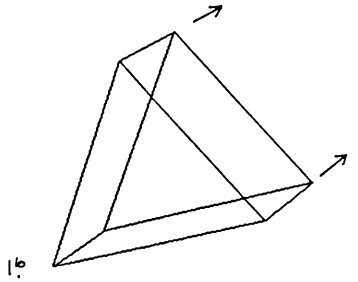
The images in the left column of the next page are drawn with a vanishing point that can be marked on the screen with an asterisk. The vanishing point was created by MAKE "VP [200 150]. The images in the right column are drawn with a more distant vanishing point that cannot be marked on the screen. It was created by MAKE "VP [800 600].

Explain the differences between these two scene sequences in words. Be specific.

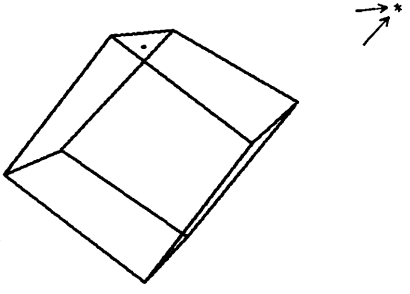
Near and distant vanishing points



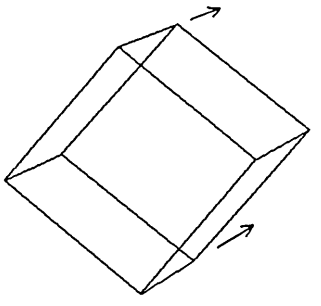
1a.



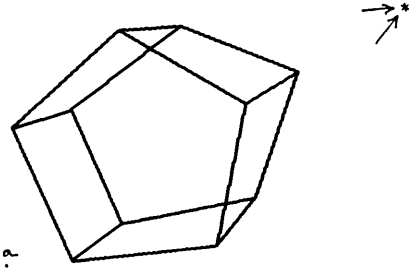
1b.



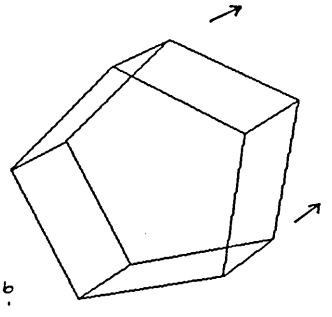
2a.



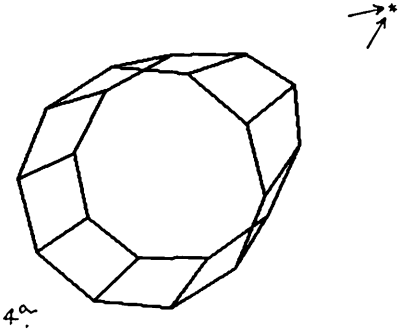
2b.



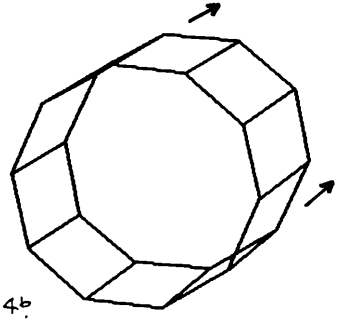
3a.



3b.



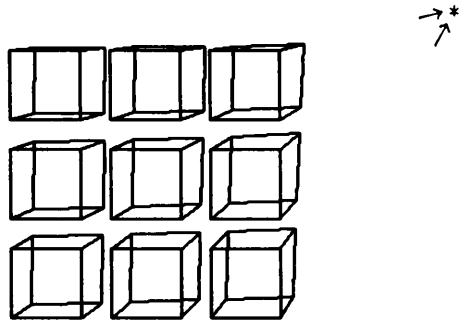
4a.

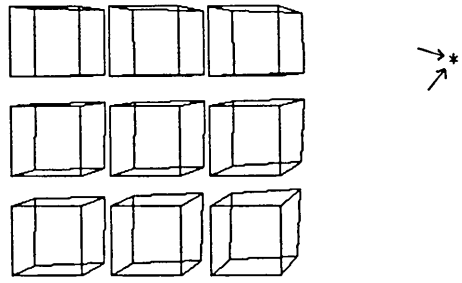


4b.

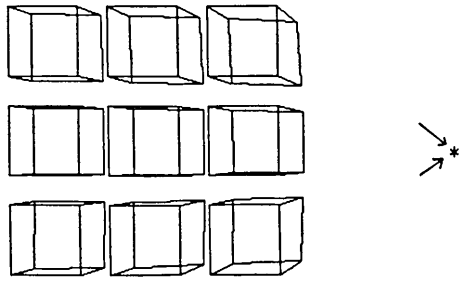
More one-point perspective experiments

Explain the following in words. How do you interpret the placement cues? How were the images created?



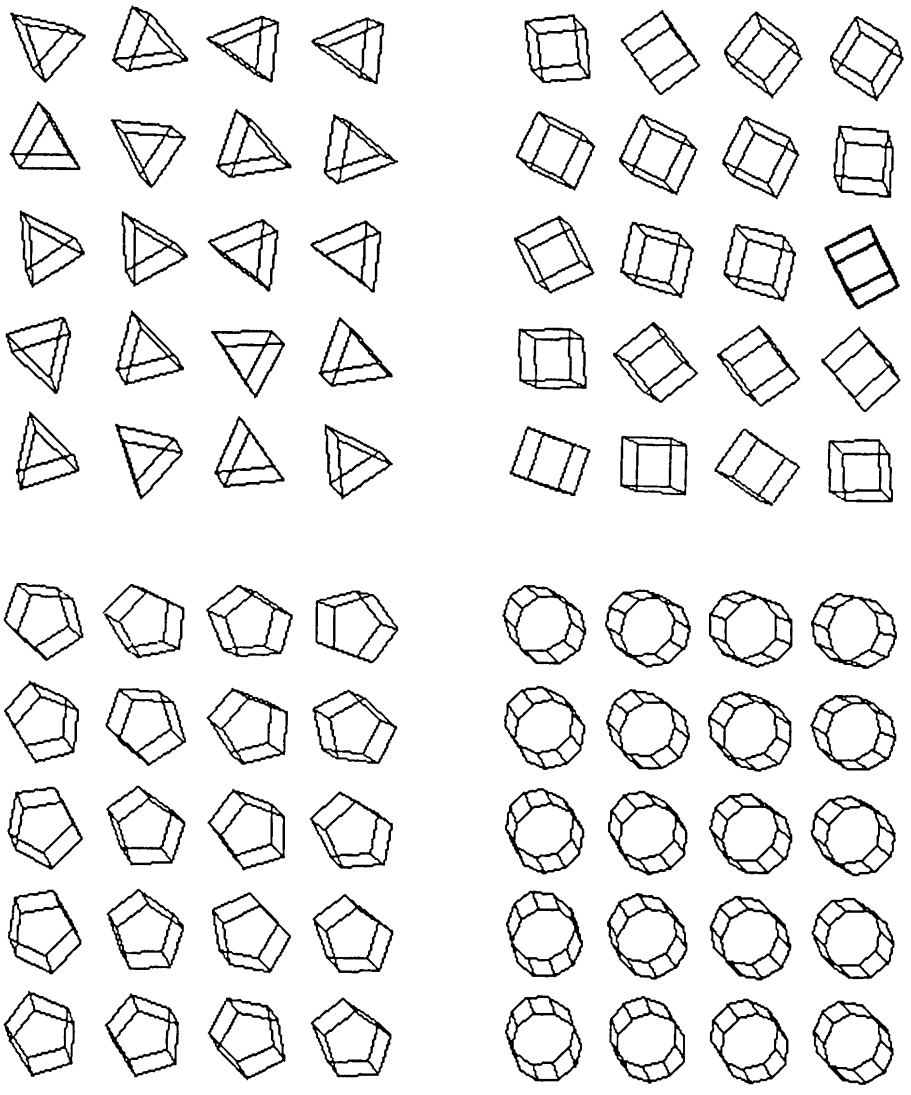


2.



3.

Tumbling NHEDRONs



NHEDRON "mistakes"

NHEDRON doesn't really draw correctly, does it? The one-point perspective system that NHEDRON uses is not classical one-point perspective. The line lengths aren't correct, and shapes near the vanishing point are badly deformed. But does this matter if you can read the placement message? NHEDRON's deformations are space data, after all, and rather more expressionistic than "correct" foreshortenings.

What is most important: to draw a scene exactly as it would look on a photograph or to express the quality of its space relationships in a way that might be *understandably original*? You know my answer. If you want a photograph of cubes, that's OK. If you want to think about cubes in space, then consider building a space-cube model with a bizarre perspective system.

Recapitulation

This chapter introduced the idea of adding depth cues to images. We modeled several varieties of such cues so that you could see how your eyes responded to them. I hope you were surprised, and pleased, to be able to watch yourself doing something about which you probably had little previous knowledge. Visual modeling can be very tricky and very revealing.

And now that you have learned something about your eye's conception of space, work with your new colleague on the exercises that follow.

Exercises

There are ten exercises for this space chapter. Be outlandish, flamboyant, and singular, but do it all multidimensionally.

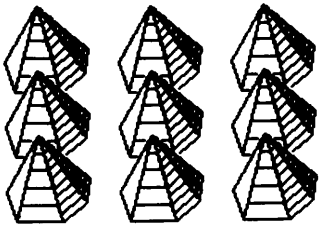
Exercise 8.1

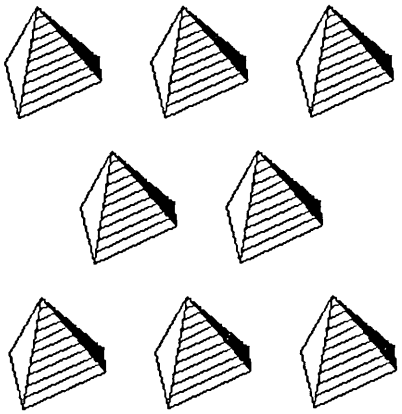
Design an Escher box-like shape that is composed of triangles. What about pyramids? A pyramid is any shape that has a polygon for its base and triangles for its sides. Build a pyramid model that will shade and place your objects in space. You may want to include the ability to overlap pyramids using the painter's algorithm. Use the model to produce a variety of pyramid graphics. Strive to make each scene as different from the others as possible, don't overlook "impossible" pyramids.

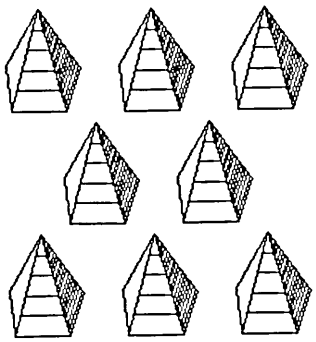
The real exercise is to explore how we see pyramid-ideas in space, so don't forget to comment—in prose—on what you have learned. Does your eye read pyramid shapes differently from the way it reads the box shapes investigated in the chapter? Why might this be so?

The following pages suggest one approach to this exercise; in fact, it's the way I did it. You might like to review my geometry, but probably you will want to model pyramids differently.

Shaded pyramids in space for Exercise 8.1







Geometry and trig for my shaded pyramids

Study the following figures. Imagine that the pyramid is hung, like a Christmas tree ornament, from its apex. See figure 1 on the next page. Image the turtle to be at position (1), pointing downward. If we can tell the turtle how to draw the leftmost triangular face of the pyramid, it will know how to draw the remaining ones.

But first, how shall we define the shape characteristics of the pyramid that we want the turtle to draw? See figure 2. I have selected the lengths of the visible edges of the pyramid (the "a"s) and the angles that separate these edges (the "α"s).

Second, how shall we indicate the kind of shading we want on each face? See figure 3. I will want shading lines that are parallel to the base of each face (the "c"s), and I will tell the turtle the number of such lines I want per face.

OK, let's get back to structuring the turtle's walk through the first face. Figure 4 shows the bits we need to relate. We know a, b and α, and we will need to calculate the rest. The page after next gets on with it. If you can't follow my method, or if you don't want to, you are ready to find your own approach. Pick whatever comes naturally to you, but try to be as simple and straightforward as possible.

Figures for my shaded pyramids

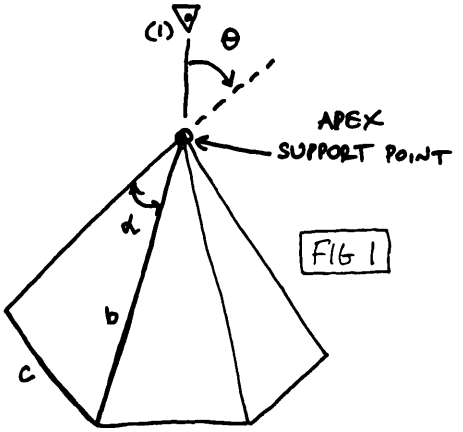


FIG 1

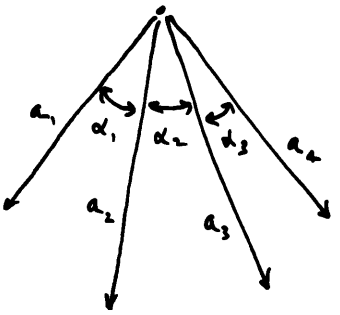


FIG 2

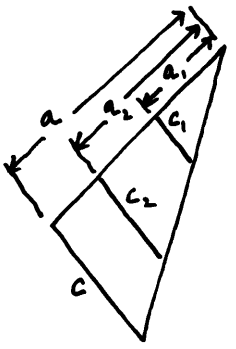


FIG 3

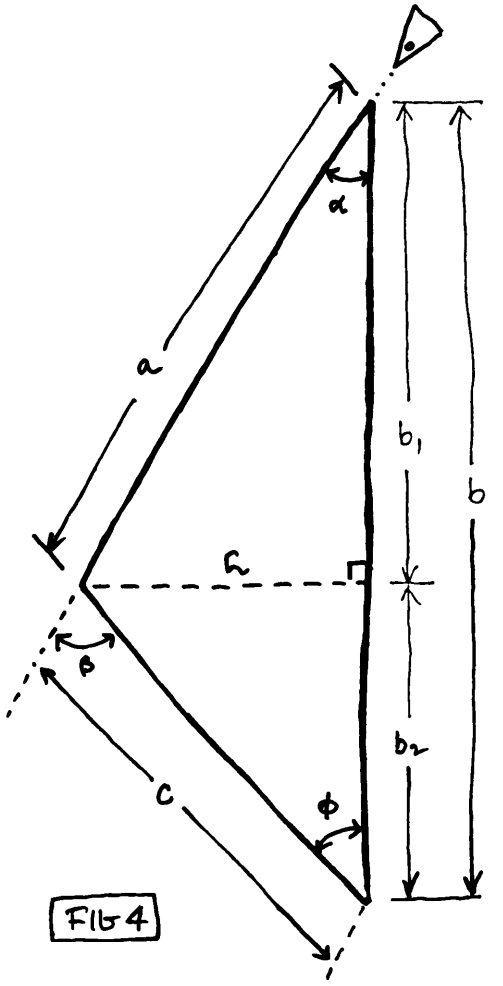


FIG 4

Calculation of the parts

Suppose we know a, b, and α. We need to calculate c and β:

1. h is the height of the triangle. It meets side b at a right angle dividing it into two segments: b1 and b2.

2. SIN α = h/a
---> h = a*SIN α

3. COS α = b1/a
---> b1 = a*COS α

4. b = b1 + b2
---> b2 = b - b1

5. TAN ϕ = h/b2
---> ϕ = arctangent h/b2

6. The interior angles of any triangle sum to 180 degrees
---> α + ϕ + (180 - β) = 180
---> α + ϕ - β = 0
---> β = α + ϕ
This gives us β <----

7. SIN ϕ = h/c
---> c = h/SIN ϕ
This gives us c <----

Now look back at figure 3. How do we calculate c1 and c2? The parallel "c"s divide the large triangular face into smaller but similar triangles. Remember the business about parts of similar triangles from geometry?

Using the idea of similar triangles, we can say that c1 is to c as a1 is to a (c1/c = a1/a). So c1 = (a1/a)*c.

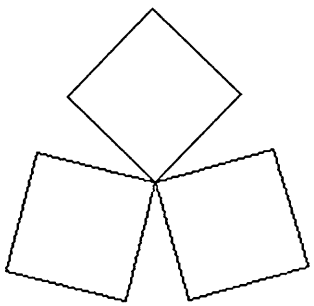
Finally, note that the turtle (already pointing downward) must turn right by the angle θ before drawing the "a" edge of the first triangular face.

Exercise 8.2

Shading techniques should be related to the shadows that different kinds of light sources would create within a scene. Try to incorporate light source arguments into an Escher box scene model. You will need to think about the placement of the light source as well as the intensity and quality (color) of the light produced.

Exercise 8.3

The Escher box motif is only one symbolic way to express a cube. There are many alternative ways, and I have included one below. Devise a number of different, visual idioms for representing cubes in space. Design a model to speak your idiom.



Exercise 8.4

Cubism was an art movement of the early twentieth century that attempted to give a visual account of the entire structure of objects in space. In practice this meant superimposing and combining several views of the object, all within a space limited by the dimensions of two-dimensional canvas. The goal was to project the idea of an object in space without being constrained to show it from a single point of view. Cubist practitioners included Picasso, Braque, and Gris.

Find a few postcard reproductions of cubist works that you like and explore the images with Logo models. Comment on how you have characterized cubist space in your work.

Exercise 8.5

What is the room that you know best? Perhaps it is your bedroom, workshop, or study; but it could also be an outdoor enclosure: a greenhouse, gazebo, or garden shed. Build a Logo model to document the shape and spatial qualities of this room. Keep your spatial vocabulary spare but expressive.

Here is the most expressive example of a personal room model that I have in my postcard collection: Van Gogh's asylum bedroom.

Exercise 8.6

Produce a shaded sphere, cylinder, or prism. Be innovative. Don't rely on the shading ideas of the chapter; invent some new ones never seen by Earthlings.

Exercise 8.7

Produce a landscape of a variety of shapes, all consistently shaded and perspected. Explain what you mean by *consistently*, though.

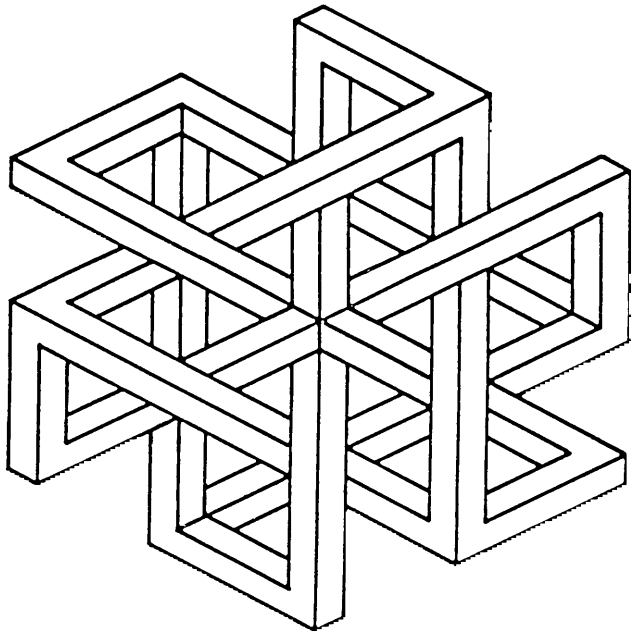
Exercise 8.8

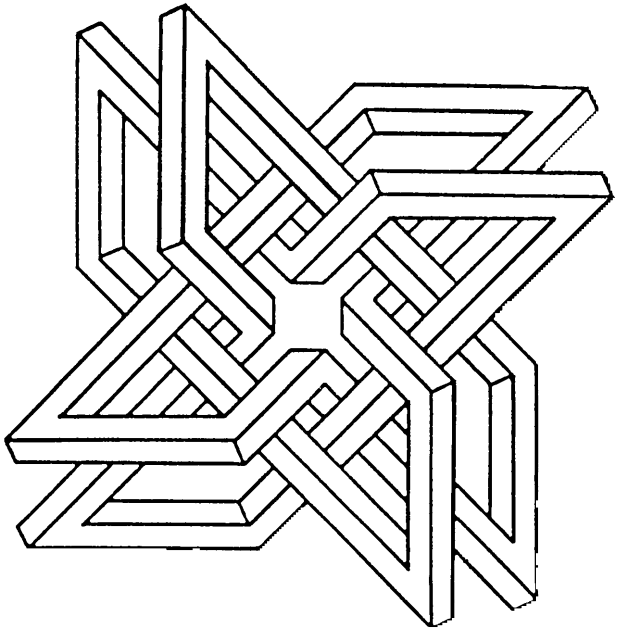
What about two- and three-point perspective? Can this be done easily with Logo? Can it be done at all with Logo? Try to explain how you would go about doing it. Then do it.

Exercise 8.9

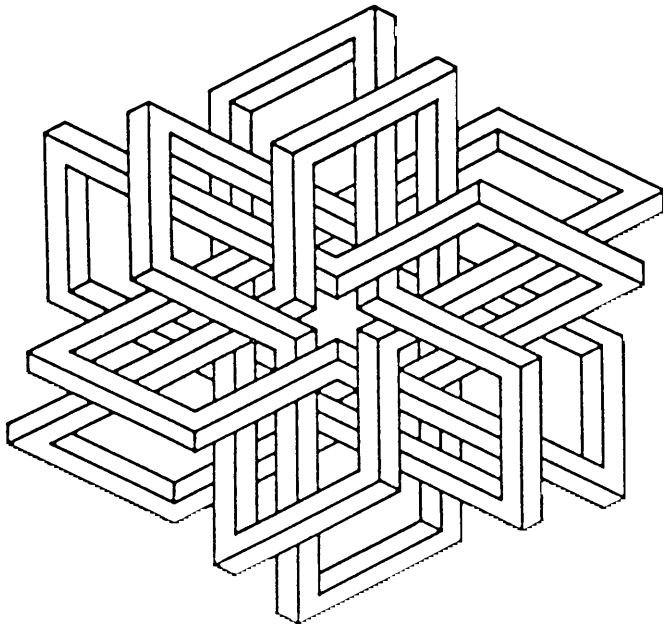
How about designing your own space? On the following pages you will see some space creations by the Hungarian artist Tamas Farkas. Don't believe the claim that they are "impossible." Compete with Farkas by designing your own impossible space grids.

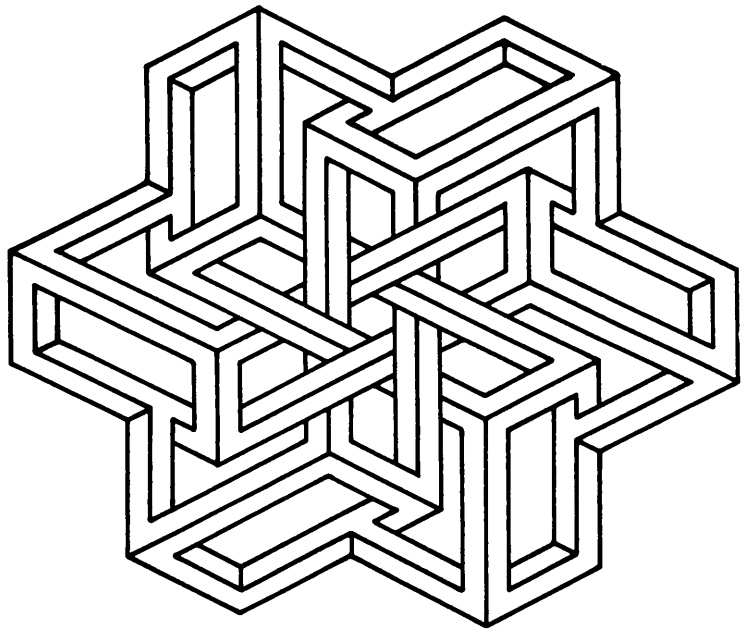
Farkas grids 1



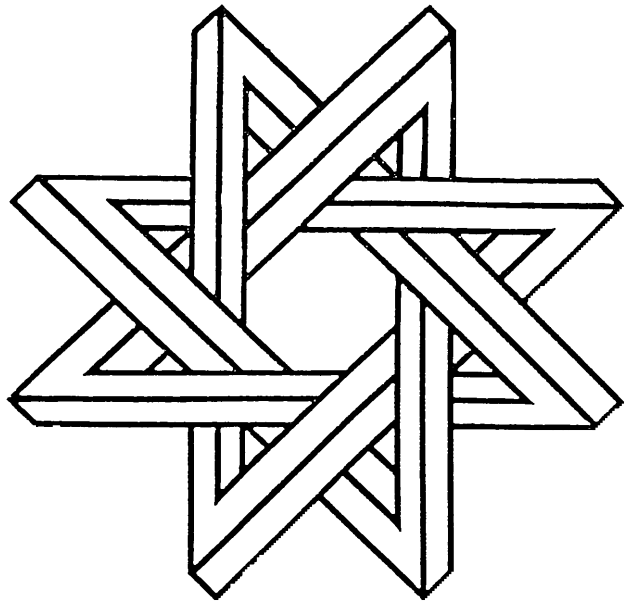


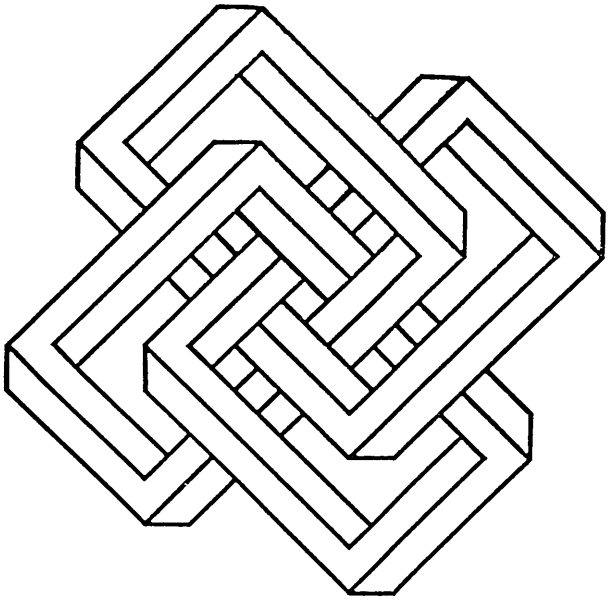
Farkas grids 2





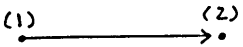
Farkas grids 3



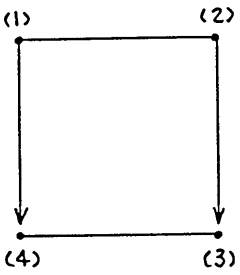


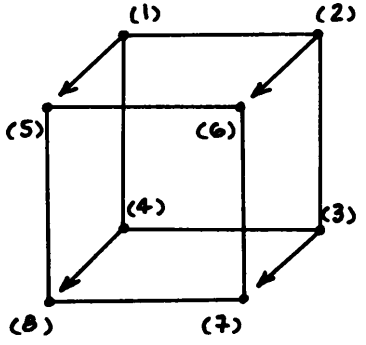
Exercise 8.10

We have drawn figures that look as if they have three dimensions. But what about four (or more) dimensions? Let's review everyday space with a few drawings. We can start with a single point: see (1) below. *No dimensions* in a single point, right? If it's very tiny. Next, draw a horizontal line from it toward (2). We have a *one-dimensional* line.



Now, pull this line downward, parallel to itself, like a window shade, toward the line (4)-(3). Connect the vertices. Now we have a square figure that is *two-dimensional*. We could call it by a fancier name, a plane, but either way, it's clearly two-dimensional. Next, let's pull this plane into three dimensions. Obviously, we must rely on some drawing conventions to force three dimensions from two-dimensional paper. We can use the conventions of this chapter to fool the eye about this. So, pull the plane, (1)-(2)-(3)-(4), "forward" into three dimensions: illustrate this by pulling it, parallel to itself, down and to the left: (5)-(6)-(7)-(8). Connect the vertices. We have a *three-dimensional* cube.

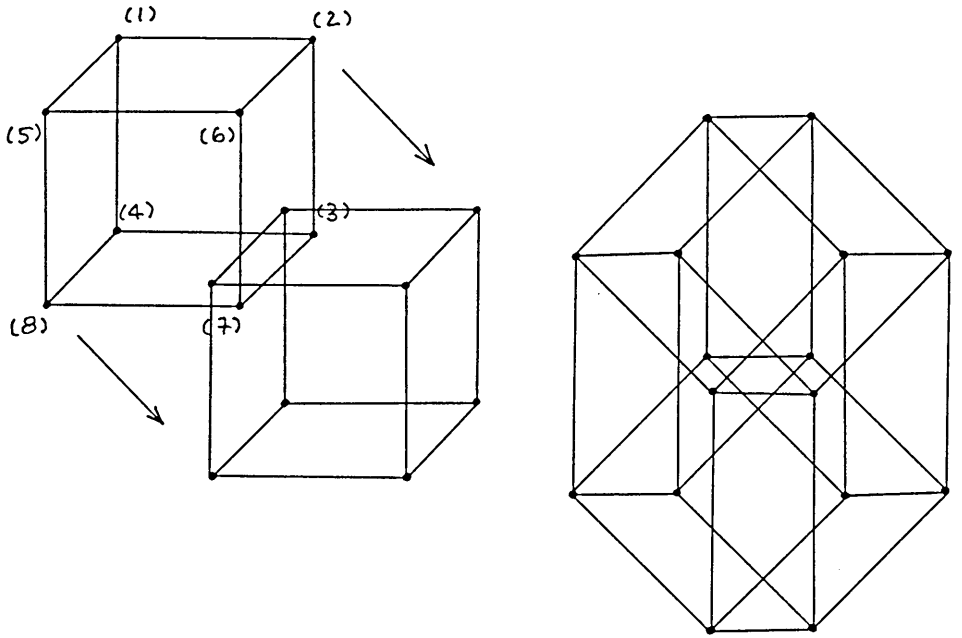




Now, for the fourth dimension. Of course this can't be done. But let's diagram the *idea* by pulling the three-dimensional shape, the cube, parallel to

itself, into another dimension. Remember that the fourth dimension must be at right angles to the other three, as each of them were to each other. OK?

We can choose whatever visual convention we want to represent all this. Let's select the convention of pulling the cube down and to the right. Why not? Spatial systems are only conventions, aren't they? We can do anything we want as long as the eye is given some depth cue so that the drawing can be read. So let's pull the cube (1)-(2)-(3)-(4)-(5)-(6)-(7)-(8) down and to the right into the stylized fourth dimension. Connect the vertices and we have what is known as a *four-dimensional* hypercube.



Play with this hypercube convention a bit. Can you invent an alternative scheme for looking into four-dimensional scenes?

What about hyperpyramids, hyperspheres, hypercylinders? Can you draw a shaded hyper-NHEDRON with nice perspective? Do some 4D modeling, but remember to state your space conventions in words so that people can read your drawings.

Chapter 9
Closure

"Purpose shapes process. Discovery depends not on special processes but on special purposes. Creating occurs when ordinary mental processes in an able person are marshaled by creative or appropriately 'unreasonable' intentions."
D. N. Perkins

Unreasonable intentions

Fifteen years ago, soon after I arrived in Paris, I signed up for a poetry workshop at the Center for Students and Artists on boulevard Raspail. There were five of us in the class—plus the teacher. I liked the woman but questioned her teaching methods. She assigned a specific exercise each week, and we were to return the next to read our solution. I found the exercises overly structured and ridiculous, and I planned to show her how silly they were by slavishly following her tasks. My results, I was convinced, would vindicate my feelings. And then she gave the "accident" assignment; I remember it vividly.

We were to write a short poem on a local accident. How trite, I thought. On the late afternoon of the "accident" class, I returned to my section of the city grimly determined to find and report on some street mishap. I wandered around my shopping spot, the rue Lepic, hoping that something would happen. And something did, horns blew, and I rushed down the street only to slip on some ice, fallen from a bin of fish. I slammed into a table of fish-covered ice: an

accidental meeting with fish. That was the title I found on the street. I went home and got on easily with the exercise.

I am no poet, but I wrote the best poetry that I have ever written in that class. The reason for my success was, I think, that I passionately sought out my own form for each given problem. Although I wasn't sure what the form would be, I knew that I would have to explore before I found it. Very unreasonable intentions.

Problem finders

Jacob Getzels and Mihaly Csikszentmihalyi studied the ways male art students at the Art Institute of Chicago explored the problems that were assigned to them; the results were published in a book called *The creative vision: a longitudinal study of problem finding in art* (John Wiley, New York, 1976). For example, these students were asked to select a set of objects from a number provided and arrange them in preparation for drawing a still life. The qualities of the drawings were compared with the methods the students used to approach and structure the exercise. The researchers followed the students for seven years after graduation to track their success. The form of their success, or lack of it, was then compared with what the authors called their "problem finding styles" while in art school. Can you guess the results?

Getzels and Csikszentmihalyi found a relationship between a student's working procedures and both the quality of his student work and his professional success seven years later. The most effective work—in art school and later—came from students who selected the most unusual objects and who spent most time arranging and manipulating these objects before they started to draw. They began their work without having any specific conception of the organizational structure they wanted to capture in their arrangement of objects; in fact, they seemed to discover a structure that they liked through handling the objects and moving them about. As their drawings progressed, the more

imaginative students tended to rearrange their objects, to introduce new elements or to eliminate others. They changed paper, switched drawing media, and changed their drawing position more often than their less successful classmates. And when they were finished, these students said that elements in their final drawings still might be changed without destroying its character.

In addition to working styles, these researchers found that a concern for fundamentals also affected creative production. The effective *problem finders*—Getzels and Csikszentmihalyi's label for those students who found and structured their own problems—tended to describe their work as being based on matters of deep personal concern to them, even when these themes were not visually present. It seemed as if the energy for the realization of the problem finders' exploratory style was funded by their passions.

Exploratory facility

I wrote this book with the conviction that a personal problem-finding ability is necessary for real creativity. But how can this ability be encouraged in the realm of visual problems when most students suffer from a lack of exploratory facility? They may have the passions, but they lack a sufficient visual vocabulary and the most basic artistic skills and will that could have helped them start fashioning a personal vision language. I am convinced that visual modeling can substitute for these lacks. How?

Visual modeling is an environment in which you can offer yourself the gift of seeing on your own terms. The attendant luxuries—design skills, rich visual word-images and metaphors—are all there, inside that space. The substance of this book is to offer you sight of this kind of environment. *But you must not stop here.* You must extend modeling into your own worlds.

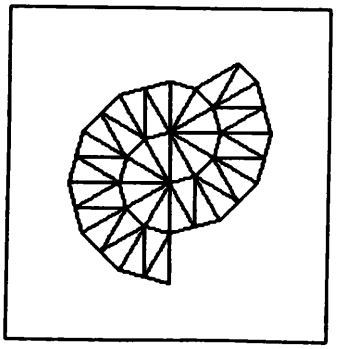
Premature closure

We haven't spoken of closure since Chapter 2. As a design spreads itself onto the surface of your screen, it may eventually repeat itself or close on top of itself. Closure may come very soon, or only after many "rotations" of your visual machinery. Be on your guard, though. Premature closure, like death and entropy, must be fought. Change your arguments, or change the models, or find another problem. But always look through closed figures; there may be others, just behind them, that are far more intriguing.

All the images in this book are now closed, or are they?

A final illustration

I wanted to end with the same simple polygons we have labored with throughout this book but in a design that did not, and would not, close. It is a nonperiodic tile pattern.



Reading list

I am sure that the character of this book would not be changed if I, like Getzels and Csikszentmihalyi's problem-finders, changed some of its elements. I feel the same about the following list of books. I present a relatively short selection

of some of the books that I found useful. Each of them encouraged me to think differently about visual modeling. Like everything else that I have offered you, it is a personal selection. I have not repeated the books I mentioned in the text.

Turtle graphics and the Logo language

To my mind, most Logo books are useless. There are three exceptions, however, that are quite wonderful—each for quite different reasons. The first uses the notion of turtle geometry to explore mathematics, while the second investigates—through Logo programming—the ideas of computer science. Both books require familiarity with mathematics and are not easy reads. However, most of your work on the exercises in this book could be aided by selected reading in the Abelson and diSessa book, while your Logo programming style would be extended and enriched by a careful study of Brian Harvey's books.

Finally, if you have never read about the philosophy behind the invention of Logo, you should read the word from Seymour Papert, the inventor himself, in the third book.

Harold Abelson and Andrea diSessa, *Turtle geometry: the computer as a medium for exploring mathematics*, The MIT Press, Cambridge, 1981.

Brian Harvey, *Computer science Logo style*, 3 volumes, The MIT Press, Cambridge, 1985, 1986, 1987.

Seymour Papert, *Mindstorms; children, computers and powerful ideas*, Basic Books, New York, 1980.

Psychology

Read anything by Rudolf Arnheim. Here is a short selection:

Visual thinking, University of California Press, Berkeley and Los Angeles, 1969.

Entropy and art, University of California Press, Berkeley and Los Angeles, 1971.

Art and visual perception: a psychology of the creative eye, University of California Press, Berkeley and Los Angeles, 1974.

The power of the center: a study of composition in the visual arts, University of California Press, Berkeley and Los Angeles, 1982.

New essays on the psychology of art, University of California Press, Berkeley and Los Angeles, 1986.

Next, a hodgepodge of titles that I found of interest. They will lead you to other sources.

Carl Jung, editor, *Man and his symbols*, Doubleday & Company, New York, 1964.

E. H. Gombrich, *Art and illusion: a study in the psychology of pictorial representation*, Princeton University Press, Princeton, New Jersey, 1960.

Ned Block, editor, *Imagery*, The MIT Press, Cambridge, 1981.

Peter van Sommers, *Drawing and cognition: descriptive and experimental studies of graphic production processes*, Cambridge University Press, Cambridge, 1984.

D. N. Perkins, *The mind's best work: a new psychology of creative thinking*, Harvard University Press, Cambridge, 1981.

Ellen Winner, *Invented worlds: the psychology of the arts*, Harvard University Press, Cambridge, 1982.

William J. J. Gordon, *Synectics: the development of creative capacity*, Harper & Row, New York, 1961.

Liam Hudson, *Contrary imaginations: a psychological study of the English schoolboy*, Methuen, London, 1966.

Mildred L. G. Shaw, *On becoming a personal scientist: interactive computer elicitation of personal models of the world*, Academic Press, London, 1980.

Design

There are scores of books on design, and many contain material of interest. I am most fond of the following.

György Kepes, editor, *Education of vision*,, George Braziller, New York, 1965.

György Kepes, editor, *Module, proportion, symmetry, and rhythm*, George Braziller, New York, 1965.

György Kepes, editor, *The nature and art of motion*, George Braziller, New York, 1965.

Johannes Itten, *Design and form: the basic course at the Bauhaus*,Van Nostrand Reinhold Company, New York, 1975.

Kenneth F. Bates, *Basic design: principles & practice*, Barnes & Noble Books, New York, 1979.

David A. Lauer, *Design basics*, second edition, Holt, Rinehart and Winston, New York, 1985.

Other design books I like:

Peter S. Stevens, *Handbook of regular patterns: an introduction to symmetry in two dimensions*, The MIT Press, Cambridge, 1984.

Charles Bouleau, *The painter's secret geometry: a study of composition in art*, Harcourt, Brace & World, New York, 1963.

Magnus J. Wenninger, *Polyhedron models*, Cambridge University Press, Cambridge, 1971.

Magnus J. Wenninger, *Spherical models*, Cambridge University Press, Cambridge, 1979.

D'Arcy Wentworth Thompson, *On growth and form*, Cambridge University Press, Cambridge, 1961.

Dover press publications are a treasure trove of design ideas. They are located at 180 Varick Street, New York, NY 10014. Write them for their catalogue.

Finally, find a library that takes the following journal. It is quirky and quite wonderful: *Leonardo: journal of the international society for the arts, sciences, and technology.*

Art history

Go to museums and galleries and buy lots of postcard reproductions.

A mixed bag of things

Judith Wechsler, editor, *On aesthetics in science*, The MIT Press, Cambridge, 1978.

Benoit Mandelbrot, *The fractal geometry of nature*, W. H. Freeman and Company, San Francisco, 1977.

Douglas R. Hofstadter, *Gödel, Escher, Bach*, Basic Books, New York, 1979.

Douglas R. Hofstadter, *Metamagical themas: questing for the essence of mind and pattern*, Basic Books, New York, 1985.

Rudy Rucker, *The fourth dimension, and how to get there*, Rider and Company, London, 1985.

William Poundstone, *The recursive universe: cosmic complexity and the limits of scientific knowledge*, William Morrow and Company, New York, 1985.

Istvan Hargittai, editor, *Symmetry: unifying human understanding*, Pergamon Press, Oxford, 1986.

Edward R. Tufte, *The visual display of quantitative information*, Graphics Press, Cheshire, Connecticut, 1983.

Christopher Alexander, Sara Ishikawa, and Murray Silverstein, *A pattern language: towns, building, and construction*, Oxford University Press, New York, 1977.

Index

The MIT Press, with Peter Denning, general consulting editor, and Brian Randell, European consulting editor, publishes computer science books in the following series:

ACM Doctoral Dissertation Award and Distinguished Dissertation Series

Artificial Intelligence, Patrick Winston and Michael Brady, editors

Charles Babbage Institute Reprint Series for the History of Computing, Martin Campbell-Kelly, editor

Computer Systems, Herb Schwetman, editor

Exploring with Logo, E. Paul Goldenberg, editor

Foundations of Computing, Michael Garey, editor

History of Computing, I. Bernard Cohen and William Aspray, editors

Information Systems, Michael Lesk, editor

Logic Programming, Ehud Shapiro, editor; Fernando Pereira, Koichi Furukawa, and D. H. D. Warren, associate editors

The MIT Electrical Engineering and Computer Science Series

Scientific Computation, Dennis Gannon, editor