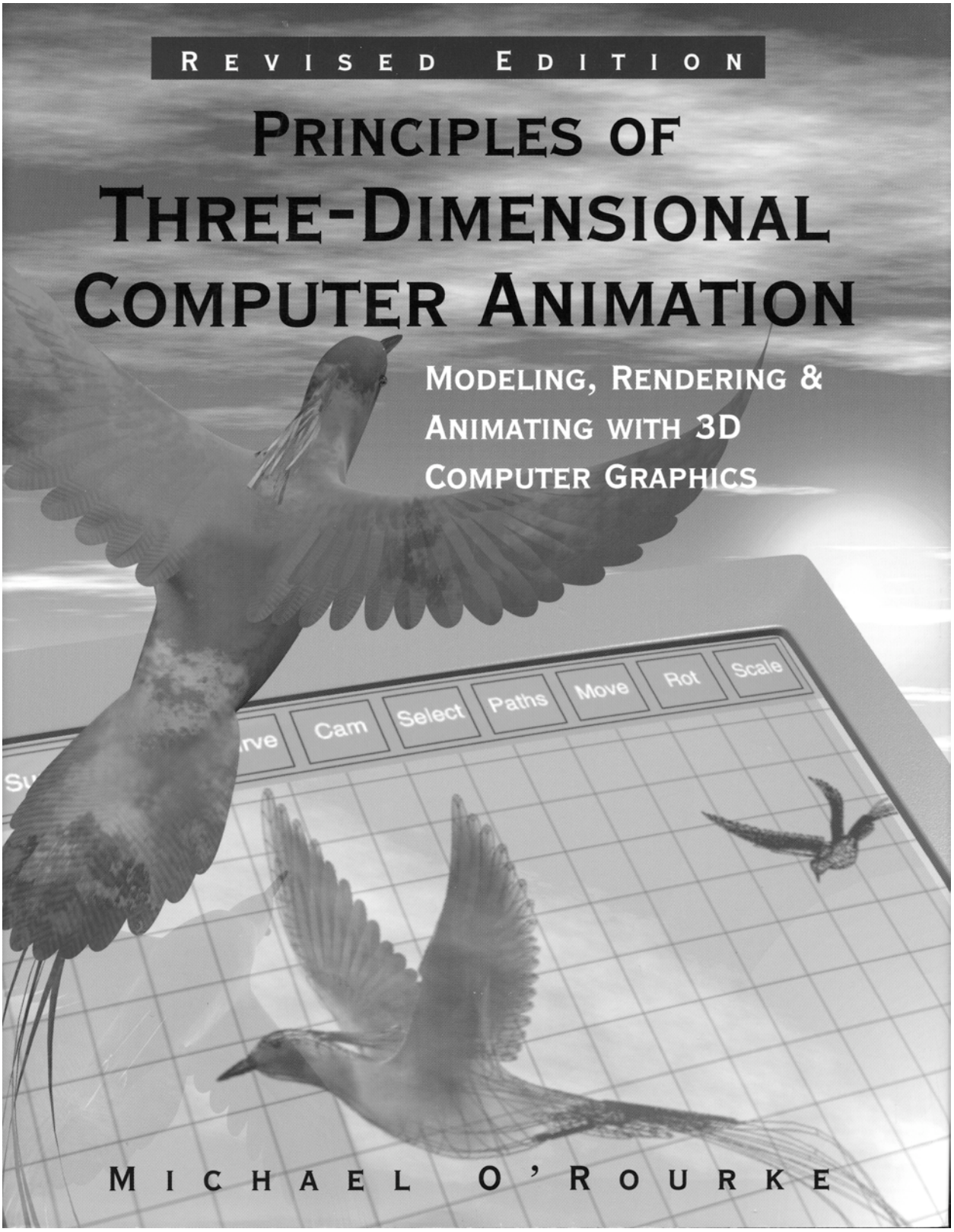

REVISED EDITION
PRINCIPLES OF THREE-DIMENSIONAL COMPUTER ANIMATION
MODELING, RENDERING & ANIMATING WITH 3D COMPUTER GRAPHICS
Cam
Select
Paths
Move
Rot
Scale
MICHAEL O'ROURKE

Just as you position the flock elements at the start of the animation to fill the starting shape, you position them at the last frame of the animation approximately to fill the ending shape (in Figure 4-54, a sphere). The starting and ending shapes determine the arrangement of the flock elements at the first and last frames, but they do not necessarily indicate how the elements move from the one to the other. By default, the elements of a flock move in an approximately straight line between the starting shape and the ending shape. Once again, the word "approximately" indicates that some degree of randomness is introduced into the path of each element. Usually, you control a randomness parameter, which increases or decreases the amount that each element deviates from the default straight-line path. If you intend the flock to move in some path other than a straight line, you can animate the location of the ending shape. This change causes the flock to "pursue" the ending shape, changing the path of the flock over the course of the animation as the flock tracks the moving target (see Figure 4-54).

Procedural Animation

Another technique offered by some systems involves a very different kind of approach to modeling and animation. In these systems, there is an intermediate step between the interactive work of modeling and animating a sequence and the final rendering of this sequence. In these systems, a text file, usually called a **scene file**, is written out by the interactive package. The scene file is a complete verbal and numerical description of everything in the scene—the models, the lights, the surface characteristics, the camera, as well as any animation that these elements might have.

A 3D software system might use a proprietary language for this scene file, or it might use a standardized language. In either case, the scene file is written in a programmable language. (Since the C programming language is most commonly used by professional graphics programmers, many scene-file languages are adapted from the C language.) By editing the text file with a word processor or text editor, you effectively can write a program to generate models and animation. Since a program is often referred to as a *procedure*, this technique is called **procedural modeling** or, when animation is involved, **procedural animation**.

A simple example of procedural modeling is use of the programmable scene language to create instances of an existing model. (See *Transformations*, in chapter 1, for a discussion of instances.) Suppose you use the standard functions of a software package to interactively model a tree. You then write out a scene file. This text file contains a description of the geometry and the surface characteristics of the tree, and these descriptions constitute the definition of the tree. You then edit the scene file, adding a loop statement that generates a number of instances of the original tree model. Each time the tree is instanced within this loop, the transformations on the tree (location, rotation, and scale) are recalculated, so that each new tree is in a different location and is of a slightly different size and orientation. Using this procedural approach, you easily can model an entire forest of trees.


```
/* define all elements of the scene */

define variables

define image size
define camera for scene
define surface characteristics
define lights

define tree model geometry

/* model a forest by instancing the tree model with
appropriate transformations */

for each tree
    {
    calculate the transformations on the tree
    instance the tree model
    }
```

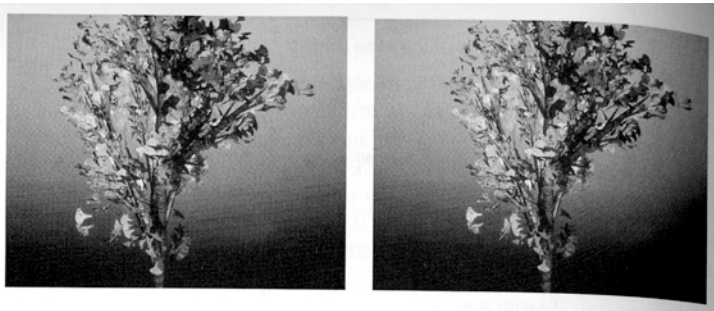
Figure 4-55. A pseudocode version of a procedure to generate a forest model. The actual code would be written in the scene-file language of the software package.

Figure 4-55 shows example of the sort of procedural modeling that might accomplish such a task. This program is written in **pseudocode**, which is an informal, Englishlike version of the logic of a program. In actual practice, you write this code in the specific scene language of the software package you use. Notice that in this pseudocode objects and variables can be defined, just as in a normal programming language, and that looping is one of the programming tools available.

Procedural-modeling applications, like particle systems (see the previous section), often make use of randomness. The forest model described above, for example, will not be very convincing if all the trees are positioned at very regular intervals, are all of exactly the same height, and so on. A certain amount of irregularity and unpredictability makes the model feel realistic.

Total randomness, however, is rarely useful. If, for example, the size of the trees is totally random, some of them might be only an inch high and some might be miles high! Clearly, this will not work. More commonly, the randomness programmed into a procedural model is constrained by certain probabilities, such as a relationship between the number of trees in the forest and the height of the forest above sea level. You also probably want the height of the trees to fall within a certain range of reasonable heights, with trees of "average" height being more likely than trees that are very tall or very short. This sort of controlled randomness in a procedure is called **stochastic modeling** and is a very powerful branch of procedural modeling.

. The components
plants were
eractively. They
omposed
within a scene file.
lants were
the same scene-
(© 1992 Sylvain



The forest example as described so far is a very elementary use of procedural modeling, in that each tree is a copy of every other tree, and only the number of trees and the transformations on the trees are produced procedurally. To make the application more interesting, the tree model itself might be produced procedurally.

One way of doing this is to model interactively the components of the tree (trunk, branches, leaves, etc.), and then use the scene-file language to write a program that procedurally constructs a tree from these components. Both of the plant models in Figure 4-56, for instance, were generated from the same scene-file program. The thickness, density, and spacing of the branches and leaves were controlled by a stochastic procedure. Though this application of procedural modeling is clearly more complex than the simple instancing of predefined trees to make a forest, the procedurally modeled trees are conceptually the same as the forest example. That is, a few model elements were modeled interactively using standard modeling techniques, then instanced procedurally within a program loop. Transformations were calculated for each instance of each element within the appropriate loop.

Figure 4-57 shows an example of what the pseudocode for such a procedural model might look like. Notice that the pattern is the same as in the pseudocode for the forest: define model geometry, loop, calculate transformations, and instance. In the more complicated procedure, however, special care must be taken with the placement of the transformations relative to one another. The model being generated by this procedure should be organized hierarchically if it is to function properly. That is, the transformations effected on the whole tree should propagate downward to all the branches of the tree; the transformations on each branch of the tree should propagate downward to all the leaves of that branch; and so on. In order to accomplish this, you use one of the standard textual representations for the hierarchical structuring of transformations (see chapter 1, *Hierarchies*). For example, curly brackets can be used to push and pop the transformations of the hierarchy.

Both of the examples discussed so far involved using procedural modeling to instance and transform preexisting geometry. However, you also can

```
/* define all elements of the scene */
    .
    .
    .
define trunk model geometry
define branch model geometry
define leaf model geometry
    .
    .
    .

/* model a tree by instancing its components with
appropriate transformations */

push /* push trunk transformations */
calculate transformatons for trunk
instance trunk

calculate number of branches
for each branch of the tree
    { /* start branch loop */

    push /* push branch transformations */
    calculate transformations for branch
    instance branch

    calculate number of leaves
    for each leaf of the branch
        { /* start leaf loop */
        push /* push leaf transformations */
        calculate transformations for leaf
        instance leaf
        pop /* pop leaf transformations */
        } /* end leaf loop */

    pop /* pop branch transformations */
    } /* end branch loop */

pop /* pop trunk transformations */
```

Figure 4-57. Pseudoc
the plant modeled pro
in Figure 4-56.

use procedural modeling to generate the geometry of a model. With this approach, you write a section of program code in the scene file to generate actual geometric data. These data might consist of a list of polygon vertices and corresponding coordinates, or perhaps, if the system is a spline modeler, a list of control points and corresponding coordinates. Using this approach you could write code to generate the geometry of the leaf, the branch, or the trunk of the tree in your forest model.

Just as a model can be defined procedurally, so too can the motion of a model be defined procedurally through the process known as **procedural animation**. Anyone familiar with basic trigonometry can program simple geometric patterns, such as circles and parabolas and sine curves. Using a few simple trigonometric functions, you can develop a scene-file program in which objects move about in these patterns. On a somewhat more complex level, you can program the growth of plants procedurally, by controlling the

number and size of the branches and leaves over time, or the density of leaf development. You also can define and animate surface characteristics and lighting for any model procedurally.

Effectively, anything that can be defined in terms of some variable or set of variables can be generated and animated in a scene-file program. In an animation of plant growth, for example, you might define the color of the leaves to progress from a lime green when the leaves are young to a deeper, richer green when they have reached maturity. In developing an animation like this, you usually have access to many of the same functions, subroutines, and programming devices that you have available in a normal programming language. Sine and cosine functions, the absolute-value function, remainder functions, various types of ease functions, loop statements, conditional statements, arrays, and so on are standard tools in most programmable scene languages.

Some of the most impressive examples of procedural modeling and animation involve very sophisticated uses of these programming techniques in conjunction with complex mathematics and physics. When scientists develop animations to simulate the orbit of a satellite around a planet, or the growth of a storm front, they use procedural modeling and animation, usually employing data from real-world measurements to produce their simulations.

The film industry produces some of its most stunning special effects through three-dimensional procedural modeling and animation as well. The water creature in *The Abyss* and the stampeding herd of dinosaurs in *Jurassic Park* are two examples. Large special-effects houses normally have both artists and programmers on staff, and when a job comes in that requires a new special effect, the artists and programmers collaborate. Based on what the artists and art directors are looking for, the staff programmers develop code to procedurally generate and animate the models involved. The artists then work with these procedures and make suggestions for changes. This process goes back and forth until the desired effect is achieved.

But what about those animators who do not have a staff of professional programmers to help develop procedural techniques? Sophisticated procedural modeling and animation can require programming skills that many artists and animators do not possess. In addition to the programming skills, these techniques require (depending on the task) a knowledge of physics, biology, meteorology, or other sciences. Clearly, not everyone possesses the scientific skills to develop sophisticated simulations or special effects. However, any animator willing to devote a little time to learning the grammar of a programmable scene language can gain access to some of the power of procedural modeling and animation.

Also bear in mind that many of the interactive techniques discussed in this book began as procedural techniques available only to programmers. It is the nature of the research-and-development process in 3D computer graphics that a technique initially can be accomplished only by those who know how to write the program code that makes it happen. Eventually, however, if the technique has wide appeal, it is written into the interactive portion of commercially available software packages and becomes available to a wider range of users, including artists and animators who may have no particular programming skills.

For example, the flying and bouncing bowling balls mentioned earlier in this chapter (color plate 2) were animated with motion-dynamics techniques. At the time this animation was produced, these techniques did not exist in any interactive software package on the market, and the artist had to write program code in a scene file to produce the motion dynamics of the balls. Today motion-dynamics techniques are a standard part of many commercially available software packages, and tomorrow they will be available in many more.