11

PROCEDURAL SYNTHESIS OF GEOMETRY

JOHN C. HART

This chapter focuses on procedural methods that generate new graphical objects by synthesizing geometry. These methods have been designed to model the intricate detail found in natural shapes like plants. Unlike the shader, which is treated as an operator that can deform the geometry of some base reference object, procedural geometry synthesis creates an entirely new object by generating its geometry from nothing.

The ideas of procedural geometry synthesis extend naturally to procedural scene synthesis. Not only are these techniques capable of growing a tree from a sprout, but they can also populate an entire forest with unique instances of the tree (see Figure 11.1) (Deussen et al. 1998). They can also be applied to other, nonbiological applications, such as the automatic synthesis of entire cities of unique buildings (Parish and Müller 2001).

The most popular method for describing procedural models of plants and natural shapes has been the L-system (Prusinkiewicz and Lindenmayer 1990). The L-system is a grammar of replacement rules that capture various self-similar aspects of biological shape and development. The grammar operates on a set of turtle graphics symbols that translate the words generated by the grammar into graphical objects. The next section briefly reviews the L-system and turtle graphics and demonstrates them with a few samples.

Algorithms for the synthesis of procedural geometry fall into two classes. Data amplification algorithms evaluate procedures to synthesize all of their geometry; lazy evaluation algorithms evaluate the geometry synthesis procedures on demand. These paradigms each have benefits and drawbacks, which we will compare later in this chapter.

While the L-system, coupled with turtle graphics, has been an effective and productive language for articulating a large variety of natural and artificial shapes and scenes, it does suffer two shortcomings.

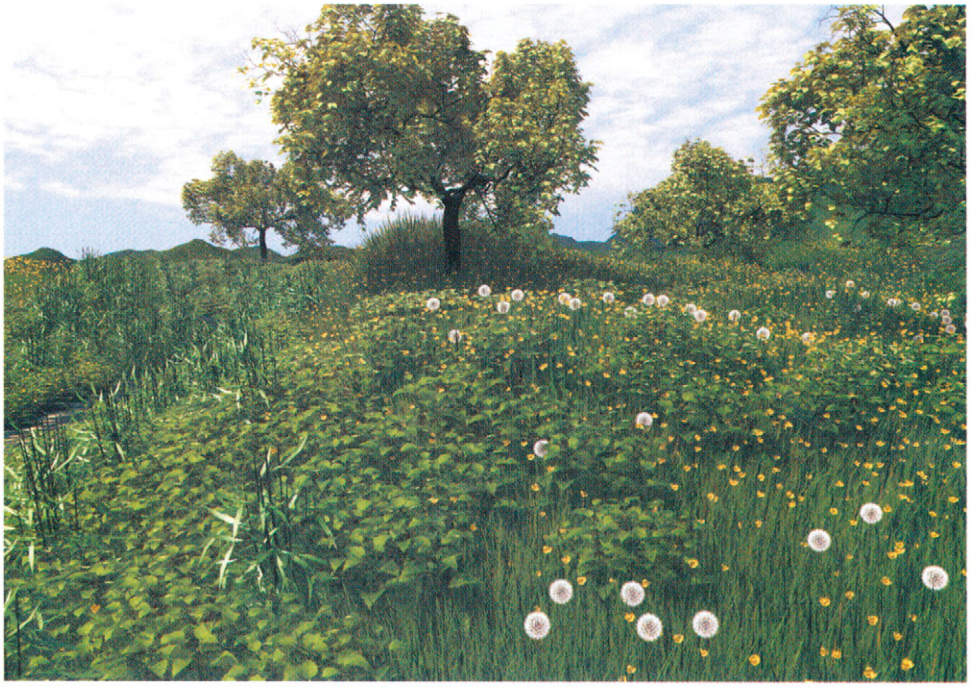

FIGURE 11.1 A procedurally synthesized environment where each plant species was ray-traced separately and composited into the final scene. The renderer executed a plant modeling program called `xfrog` to generate the plants on demand, using a lazy evaluation technique described later in the chapter. Image courtesy of Oliver Duessen. Copyright © 1998 Association for Computational Machinery, Inc.

Procedural techniques are used to model detail, but the efficient processing and rendering of scenes containing a large amount of geometric detail require it to be organized into a hierarchical, spatially coherent data structure. Turtle graphics objects are specified by a serial stream of instructions to the turtle, which, in many cases, is not the most effective representation for processing and rendering.

Furthermore, L-systems are difficult to process by humans. The L-system works with a large variety of symbols to specify operations within a full-featured 3D turtle-based scene description language. While these symbols are mnemonic, they can be likened to an assembly language. Their density makes them difficult to visually parse, and the large number of symbols used makes their articulation cumbersome.

The scene graph used by most commonly available modeling systems can be used to perform some limited procedural geometry synthesis. The replacement rules in an L-system that allow it to model self-similar detail can be represented in the scene graph using instancing. We will investigate the use of the scene graph for procedural geometry synthesis and demonstrate its abilities on a few simple procedural models.

The scene graph has some major benefits as a technique for procedural geometry synthesis. Its models can more easily be organized into hierarchical, spatially coherent data structures for efficient processing and rendering. Moreover, the articulation of scene graphs is more familiar to the computer graphics community, as demonstrated by the variety of scene description languages and programming libraries based on it.

But the standard form of the scene graph available in graphics systems is too limited to represent the wide variety of shapes and processes modeled by L-systems. We will show that an extension of the capabilities of the scene graph, called *procedural geometric instancing,* allows it to represent a larger subset of the shapes that can be modeled by an L-system. This subset includes a wide variety of the natural detailed shapes needed by typical production graphics applications.

THE L-SYSTEM

The L-system is a string rewriting system invented by Aristid Lindenmayer (1968). It was later shown to be a useful tool for graphics by Alvy Ray Smith (1984). Przemyslaw Prusinkiewicz then led an effort to develop it into a full-featured system for modeling the behavior of plant growth (Prusinkiewicz and Lindenmayer 1990).

An L-system is a grammar on an alphabet of symbols, such as "F", "+", and "-". The development of an L-system model is encapsulated in a set of *productions*—rules that describe the replacement of a nonterminal symbol with a string of zero or more symbols. For example, a production might be

$$\texttt{F} \rightarrow \texttt{F+F--F+F} \tag{11.1}$$

which says that every F in the input string should be replaced by F+F--F+F, which increases the number of nonterminal symbols in the string, causing it to grow with each production application.

This L-system growth is seeded with an *axiom*—an initial string such as the singleton F. Applying the production turns the initial string into F+F--F+F. Applying it again to this result yields

```
F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F
```

with spaces inserted to aid readability.

Unlike context-free grammars, L-system productions are applied in parallel, replacing as many symbols as possible in the current result. Hence the output of an L-system does not depend on the order the productions are applied.

This is a deterministic context-free L-system (a.k.a. a D0L-system), because each nonterminal symbol is the trigger of only one production, and this production does not care what symbols are to the left or right of the nonterminal symbol. An L-system can also be context-sensitive such that a nonterminal symbol would be replaced by something different depending on its surrounding symbols.

The symbols that an L-system operates on are assigned a geometric meaning. One such assignment is to interpret the strings as instructions to a turtle impaled with a pen (Abelson and diSessa 1982). The symbol "F" moves the turtle forward (leaving a trail) and the symbol "+" (respectively "-") rotates the turtle counterclockwise (clockwise) by a preset angle. Using the results of the example L-system and setting the angle of rotation to ± 60° yields the shapes shown in Figure 11.2.

In this manner, we can make a large variety of squiggly lines. We want to use the L-system, however, to model plant structures. Plant structures use branching as a

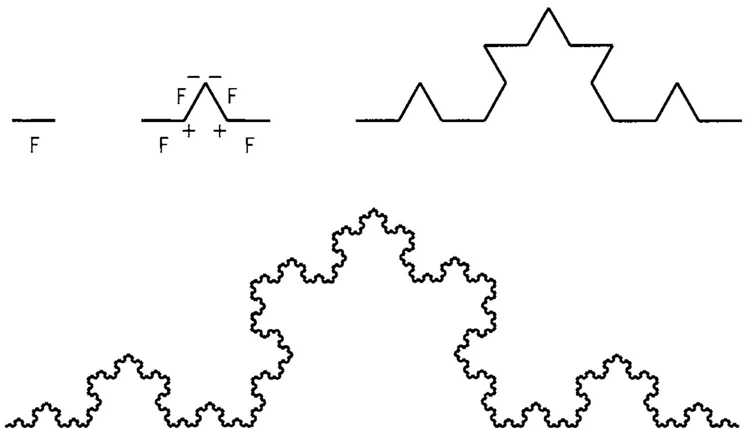

FIGURE 11.2 The von Koch snowflake curve development modeled by the L-system (11.1) and the shape to which it eventually converges.

means of distributing leaves that collect photosynthetic energy across a wide area, while concentrating the flow of nutrients to and from a similarly branching root system through a single trunk.

When a sequence of symbols generated by the L-system completes the description of a branch, the turtle must return to the base of the branch to continue describing the remainder of the plant structure. The L-system could be cleverly designed to allow the turtle to retrace its steps back to its base, but it would be cumbersome, inefficient, and incomprehensible. A better solution is offered through the use of brackets to save and restore the state of the turtle. The left bracket symbol "[" instructs the turtle to record its state (position and orientation); the right bracket "]" instantly teleports the turtle to its most recently stored state. This state information can be stored on a stack, which allows bracket symbols to be nested.

These state store/restore commands allow L-systems to model branching structures. For example, the L-system

$$\texttt{F} \rightarrow \texttt{F[+F]F[-F]F} \tag{11.2}$$

grows the sprout F into the trees shown in Figure 11.3.

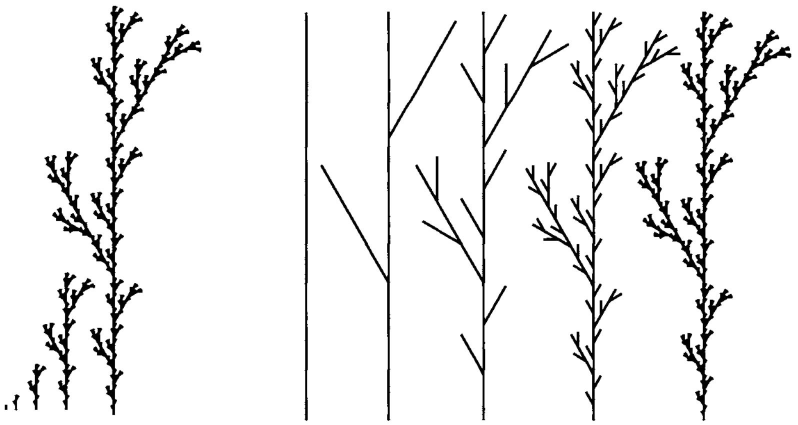

FIGURE 11.3 Development of the simple tree modeled by the L-system (11.2). (a) In this progression, the segment length remains constant, which simulates the growth of the bush from a sapling. (b) In this progression, the length of the segments decreases by 1/3 in each iteration and the tree increases its detail in place. While the progression in (a) is a developmental model useful for simulating the biological and visual properties of the growth of the tree, the progression in (b) represents a hierarchical model useful in efficiently processing and rendering the tree.

The turtle graphics shape description has been extended into a full-featured three-dimensional graphics system. In three dimensions the symbols "+" and "-" change the turtle's *yaw,* and the new symbols "^" and "&" change the turtle's pitch (up and down), whereas "\" and "/" change its roll (left and right). We can use these 3D turtle motions to grow a ternary tree with the L-system production

```
F → F[&F][&/F][&\F]
```

which attaches three branch segments to the end of each segment, shown in Figure 11.4 along with the resulting tree after several iterations of the production.

In these examples, the segment lengths and rotation angles have all been identical and fixed to some global constant value. When modeling a specific growth structure, we may want to specify different segment lengths and rotation angles. The elements of the turtle geometry can be parameterized to provide this control. For example, `F(50)` draws a segment of length 50 units, and `+(30)` rotates the turtle 30° about its yaw axis. Note that using a 3D turtle, Figure 11.2 can be constructed by the production

```
F → /(180)-(30)F+(60)F-(30)\(180)
```

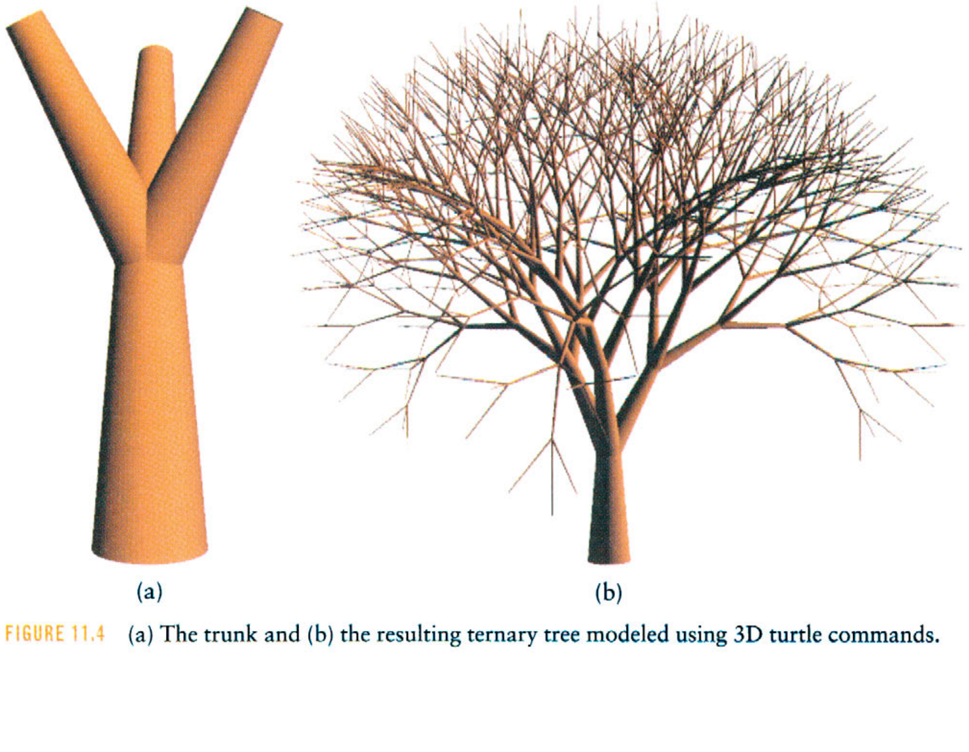
(a) (b)

FIGURE 11.4 (a) The trunk and (b) the resulting ternary tree modeled using 3D turtle commands.

which constructs the snowflake curve out of two smaller inverted snowflake curves. The curves are inverted by flipping the turtle over, which exchanges counterclockwise with clockwise.

In three dimensions, the line segments are replaced with cylinders. These cylinders represent the branches, twigs, and stems of the plant structure. Newer branches should be thinner than older branches near the base. The symbol "`!`" decreases the width of all segments specified after it in a string. Note that the "`]`" bracket resets the branch diameter to the state corresponding to the location of the "`[`" symbol in the string. Hence "`F[!+F]F`" draws a trunk of two equal-diameter segments with a thinner segment branching off its side. The segment length can also be specified directly. The "`!(`*d*`)`" substring sets the current segment diameter to *d*.

The L-system approximates the shape of the plant support structure as a sequence of straight cylinder segments. These cylinder segments can be joined by placing a sphere at the joints. The turtle command "`@O`" draws a sphere around the current turtle position whose diameter is equal to the current segment diameter. Hence "`F@O+F`" uses a sphere as an elbow between the two cylinders. Alternatively, a generalized cylinder can be constructed as a tube around a space curve by using the symbol "`@Gc`" to specify the current turtle position as a control point, and using the symbols "`@Gs`" and "`@Ge`" to start and end the curve segment. The diameter of the generalized cylinder can also be set differently at each control point using the "`!`" symbol before each "`@Gc`" substring.

The turtle geometry system quickly ran out of symbols from the ASCII character set and so needs multicharacter symbols. These multicharacter symbols are usually prefixed by the "`@`" character.

Polygons can also be defined using the turtle geometry. The turtle motion inside the delimiters "`{`" and "`}`" is assumed to describe the edges of a filled polygon. Within these braces, the symbol "`F`" describes an edge between two vertices. That operation can be decomposed into the "`.`" symbol, which defined a vertex at the current turtle position, and the "`G`" symbol, which moves the turtle without defining a vertex.

Bicubic patches can also be modeled. The substring "`@PS(`*n*`)`" defines a new bicubic patch labeled by the integer *n*. The substring "`@PC(`*n,i,j*`)`" defines the position of the (*i,j*) control point to the current turtle position. The substring "`@PD(`*n*`)`" indicates that patch *n* should be drawn.

Color and texture can also be specified. Colors are predefined in a color map, and the symbol "`;(`*f,b*`)`" defines the frontside color to index *f* and the backside color to index *b*. The symbol "`;`" alone increments the current color index. For example, a three-element color table containing brown, yellow, and green can be used for the colors of branches, twigs, and stems in the string "`F[;+F[;+F]`

[;-F]][;-F[+F][-F]]". Note that the color index is reset by the "]" closing bracket. The symbol "@T(*n*)" can be used to indicate that the output of subsequent turtle commands should be texture mapped. The parameter *n* indicates which texture map should be used. Texture mapping is turned off by the "@T(0)" substring.

The command "@L(*label*)" prints the label at the current turtle location, which is useful for debugging or annotation. The command "@S(*command*)" executes an operating system command, which could be used, for example, to provide audio feedback during the L-system processing.

PARADIGMS GOVERNING THE SYNTHESIS OF PROCEDURAL GEOMETRY

Many different interactive modeling systems tend to follow the same standard flow of data (e.g., Figures 1.5, 3.2, and 7.3 of Foley et al. 1990). The *user* articulates a conceptual model to the modeler. The *modeler* interprets the articulation and converts it into an intermediate representation suitable for manipulation, processing, and rendering. The *renderer* accepts the intermediate representation and synthesizes an image of the object. The user then observes the rendered model, compares it to the conceptual model, and makes changes according to their difference.

The procedural synthesis of geometry follows this same data flow model, although the implementation of the specifics of the intermediate representation can follow two different paradigms: data amplification or lazy evaluation.

Data Amplification

The L-system describes procedural geometry following the paradigm of a *data amplifier,* as shown in Figure 11.5. Smith (1984) coined this term to explain how procedural methods transform the relatively small amounts of information in a procedural model articulation into highly detailed objects described by massive amounts of geometry. (The figure's representation of the data amplifier with an op-amp circuit is not far fetched. The electronic amplifier scales its output voltage and

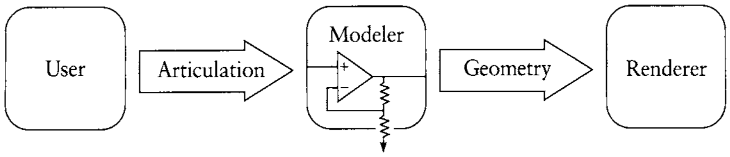

FIGURE 11.5 The "data amplifier" procedural modeling system.

then combines it with the input voltage. Such rescale-and-add processes form the basis of a variety of procedural models.)

Procedural modelers that follow the data amplification paradigm synthesize some kind of intermediate geometric representation. This intermediate representation is a scene description consisting of polygons or other primitives. For an L-system model of a tree, the intermediate representation would consist of cones representing the branches and polygonal meshes representing the leaves. The intermediate representation is then passed to a renderer to convert it into an image.

Procedural methods are used to synthesize detailed scenes. The intermediate representation for such scenes can become extremely large, growing exponentially in the number of L-system production applications. For example, the L-system description for a single poplar tree was only 16 KB, but when evaluated yielded a 6.7 MB intermediate representation (Deussen et al. 1998). Data amplification causes an intermediate data explosion as a compact procedural model is converted into a huge geometric representation, which is then rendered into an image.

The intermediate representation generated by an L-system is also largely unorganized. The geometry output by evaluating an L-system is in a stream order that follows the path of the turtle. The bracketing used to indicate branching structures does organize this stream into a postorder traversal, but this organization has not been utilized. The lack of hierarchical organization prevents renderers from capitalizing on spatial coherence when determining visibility. Standard methods can process the unorganized geometry into more efficient data structures such as a grid or octree. However, these data structures further increase the storage requirement.

We can avoid storage of the intermediate geometry representation by rendering the primitives generated by the turtle as soon as they are generated. This avoids the intermediate data explosion, but at the expense of the extra time required to manage the turtle during the rendering process. This also limits the rendering to per-polygon methods such as Z-buffered rasterization and would not work for per-pixel methods such as ray tracing.

Reeves and Blau (1985) rendered the turtle graphics generated by an L-system using particles instead of geometry. The particles were rendered directly to the screen as filtered streaks of shading. The drawback of rendering particle trails directly to the frame buffer is indirectly due to aliasing. The streaks of shading resulting from a rendered particle trail represent fine details, such as pine needles or blades of grass. These details often project to an area smaller than the size of a pixel and use the alpha channel to contribute only a portion of the pixel's color. For this technique to work correctly in a Z-buffered rendering environment, the particle traces must be rendered in depth order, and the particle traces must not intertwine. Fortunately, a

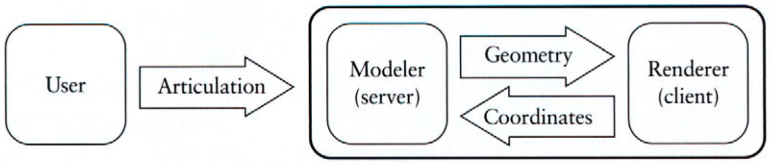

FIGURE 11.6 The "lazy evaluation" procedural modeling system.

precisely correct rendering is often not necessary. Minor occlusion errors are easily obfuscated by the overwhelming amount of visual detail found in procedurally synthesized scenes. "The rich detail in the images tends to mask deviations from an exact rendering" (Reeves and Blau 1985, p. 313).

Another option is to rasterize the streaks of shading generated by the particle traces into an intermediate volume (Kajiya and Kay 1989). Direct volume-rendering techniques can be used to render the intermediate representation to more accurately determine visibility. This technique was used to simulate the shading of fur and carpeting, and modeled a convincing teddy bear.

Lazy Evaluation

Lazy evaluation avoids intermediate geometry representation problems of the data amplifier by executing the geometry synthesis procedure only when it is needed. Lazy evaluation is illustrated diagrammatically in Figure 11.6.

Lazy evaluation facilitates a client-server relationship between the modeler and the renderer, allowing the modeler to generate only the geometry needed for the renderer to draw an accurate picture. This process saves time and space by avoiding the need to store and process a massive intermediate geometric representation.

In order for lazy evaluation to work in a procedural geometry system, the renderer needs to know how to request the geometry it needs, and the procedural model needs to know what geometry to generate. This process creates a dialog between the modeler and the renderer.

Procedural geometries can sometimes be organized in spatially coherent data structures. Examples of these data structures include bounding volume hierarchies, octrees, and grids. Such data structures make rendering much more efficient. The ability to cull significant portions of a large geometric database allows the renderer to focus its attention only on the visible components of the scene. Techniques for capitalizing on spatial coherence data structures exist for both ray tracing (Rubin and Whitted 1980; Kay and Kajiya 1986; Snyder and Barr 1987) and Z-buffered rasterization (Greene and Kass 1993).

These spatial coherence data structures also support lazy evaluation of the procedural model. For example, the modeler can generate a bounding volume and ask the renderer if it needs any of the geometry it might contain. If the renderer declines, then the procedural model does not synthesize the geometry inside it. The tricky part of implementing the lazy evaluation of a procedure is determining the bounding volume of its geometry without actually executing the procedure.

L-systems alone do not contain the necessary data structures to support lazy evaluation. The next section describes how L-systems can be simulated with scene graphs, which include the data structures and algorithms that support lazy evaluation.

THE SCENE GRAPH

Geometric objects are often constructed and stored in a local "model" coordinate system specific to the object. These objects may be placed into a scene by transforming them into a shared "world" coordinate system. The placement of a model into a scene is called *instancing*. The model is called the *master*, and the transformed copy placed in a scene is called an *instance* (Sutherland 1963).

In order to manage complex scenes containing many objects, objects can be organized into hierarchies. Instances of individual objects can be collected together into a composite object, and the composite object can be instanced into the scene. The structure of this hierarchy is called a *scene graph*.

There are a variety of libraries and languages available to describe scene graphs. For example, the Java3D and Direct3D graphics libraries contained a "retained mode" scene graph that allowed entire scenes to be stored in memory and rendered automatically. The VRML (Virtual Reality Modeling Language) is a file representation of a scene graph that can be used to store a scene graph as a text file to be used for storage and communication of scenes.

An instance is a node in a scene graph consisting of a transformation, a shader, and a pointer to either another scene graph node or a geometric primitive, as shown in Figure 11.7(a). The transformation allows the instance to be transformed by any affine transformation, and the shader allows the instances of an object to have different appearance properties. Instancing can turn a tree-structured scene graph into a directed acyclic graph, such that a node may have more than one parent in the hierarchy, as shown in Figure 11.7(b).

In the following examples, we use a specialized scene description language that encourages instancing. Scene graph nodes are named and can be accessed by their given name. An object corresponding to a scene graph node is described using the syntax

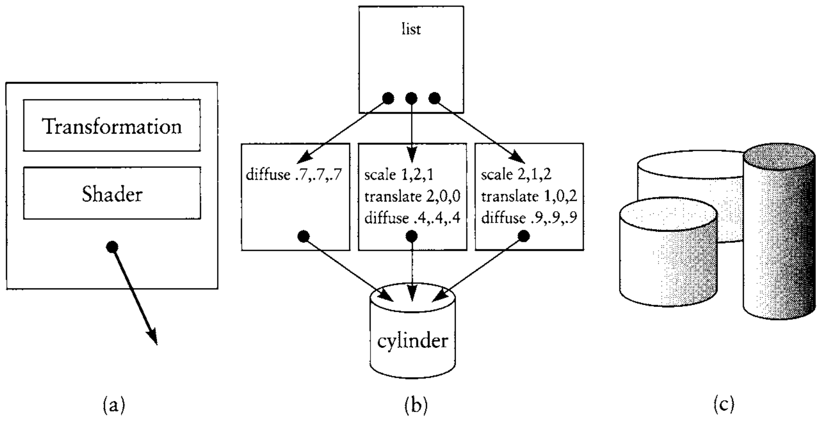

FIGURE 11.7 Anatomy of an instance: (a) The instance is a pointer to another object, with additional transformation and shading parameters. (b) The scene graph is a collection of instances, in this case a list of three instances of a cylinder primitive. (c) The configuration of the cylinders.

```
define <name> <description> end
```

where `<description>` is a list of instances and/or primitives. Instances and primitives not contained as named nodes are assumed to be elements of an implicit list node at the top of the scene graph.

Each instance or primitive precedes a list of zero or more transformation and shading commands. The transformation and shading commands are accumulated in order and act only on the most recently specified instance or primitive. When a new instance or primitive is declared, the transformation is reset to the identity, and the shading is restored to the default shading of the current parent object. Hence the specification

```
cylinder(10,1) scale 2
cylinder(10,1) rotate 30,(0,0,1)
```

creates the union of a cylinder along the y-axis of length 10 and radius 1 uniformly scaled by two, and a second cylinder also of length 10 and radius 1 rotated by 30° about the z-axis. The second cylinder is not scaled by two.

The transformation and shading commands of an instance are applied relative to the current transformation and shading of the context into which they are instanced. For example, the description

```
define A
    sphere translate (2,0,0) diffuse 1,(1,0,0)
end

define B
    A rotate 90,(0,0,1) diffuse 1,(0,0,1)
end
```

defines object A as a red unit sphere centered at the point (2,0,0), and object B as a blue sphere centered at the point (0,2,0).

The scene graph can use instancing to implement turtle graphics. For example, the turtle graphics stream "`F[+F]F[-F]F`" can be implemented in our scene description language as

```
define F
    cylinder(1.0,0.1)
end

define +F
    F rotate 30,(0,0,1)
end

define -F
    F rotate -30,(0,0,1)
end

F scale 1/3
+F scale 1/3 translate (0,1/3,0)
F scale 1/3 translate (0,1/3,0)
-F scale 1/3 translate (0,2/3,0)
F scale 1/3 translate (0,2/3,0)
```

One problem worth noting is that the state after the turtle has moved and must be maintained. This is why the translation commands must be inserted after some of the instances.

We can also use instancing to implement the productions of an L-system (Hart 1992). Consider the single-production L-system

$$\texttt{A} \rightarrow \texttt{F[+F]F[-F]F} \tag{11.3}$$

with the axiom "`A`". We can represent this structure as a scene graph as

```
define A
    F scale 1/3
    +F scale 1/3 translate (0,1/3,0)
    F scale 1/3 translate (0,1/3,0)
    -F scale 1/3 translate (0,2/3,0)
    F scale 1/3 translate (0,2/3,0)

end

A
```

where objects "F","+F", and "-F" are defined as before. An equivalent scene graph is shown in Figure 11.8(a), which yields the arrangement of cylinders shown in Figure 11.9(a).

This is even more powerful when we represent an additional iteration of development as the L-system

$$\texttt{A} \rightarrow \texttt{F[+F]F[-F]F} \tag{11.4}$$

$$\texttt{B} \rightarrow \texttt{A[+A]A[-A]A} \tag{11.5}$$

with the axiom "B". The corresponding scene graph is now

```
define +A A rotate 30,(0,0,1) end
define -A A rotate -30,(0,0,1) end

define B
    A scale 1/3
    +A scale 1/3 translate (0,1/3,0)
    A scale 1/3 translate (0,1/3,0)
```

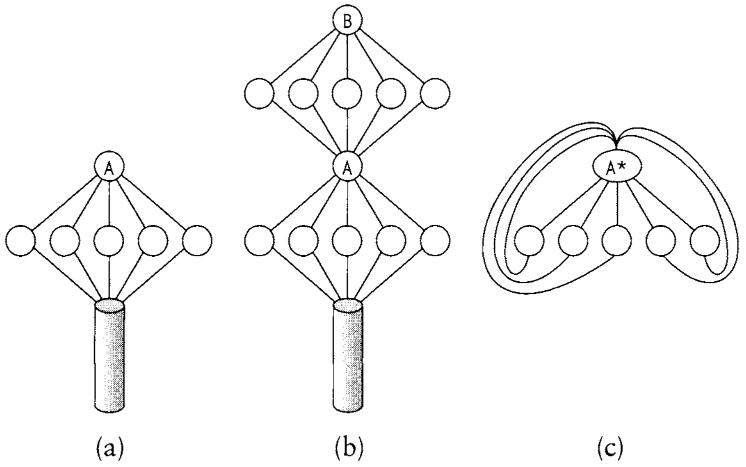

FIGURE 11.8 Scene graphs for the bush L-system: (a) one iteration, (b) two iterations, and (c) infinitely many iterations.

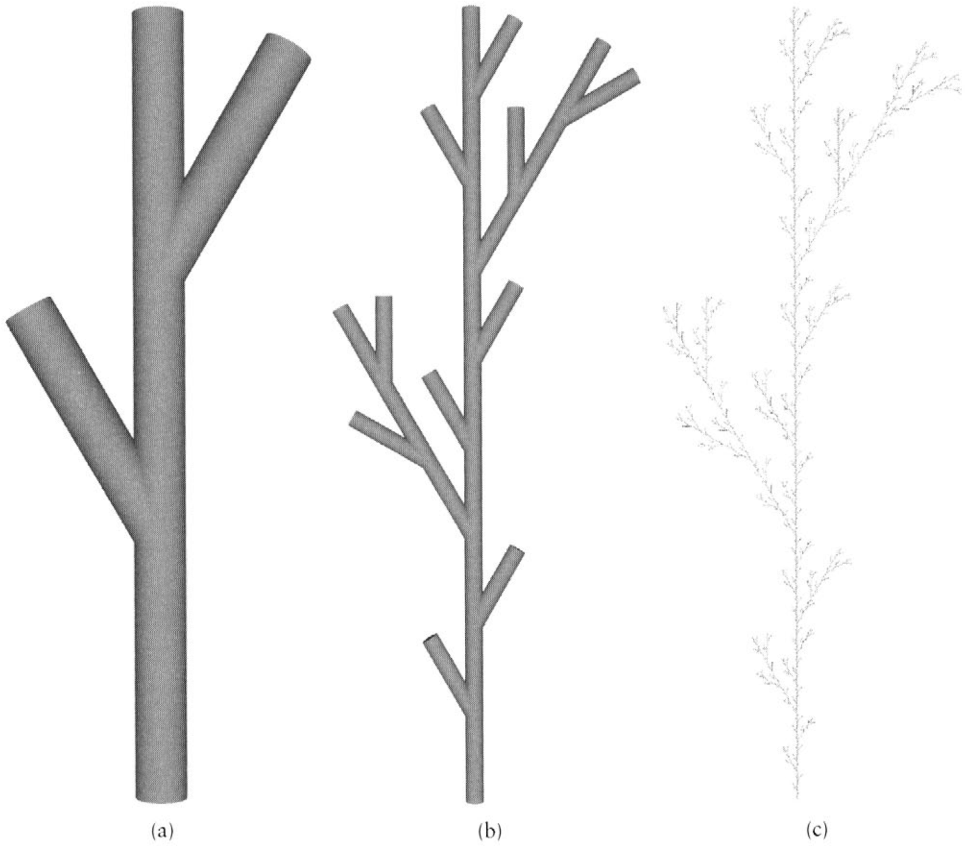
(a) (b) (c)

FIGURE 11.9 Bushes modeled by (a) "A", (b) "B", and (c) "A*".

```
    -A scale 1/3 translate (0,2/3,0)
    A scale 1/3 translate (0,2/3,0)
end

B
```

where "A" is defined as before. This scene graph is equivalent to the one shown in Figure 11.8(b) and yields the arrangement of cylinders shown in Figure 11.9(b).

Noticing the similarities between "A" and "B", you might be tempted to develop the instance

```
define +A* A* rotate 30,(0,0,1) end
define -A* A* rotate -30,(0,0,1) end
define A*
    A* scale 1/3
    +A* scale 1/3 translate (0,1/3,0)
    A* scale 1/3 translate (0,1/3,0)
    -A* scale 1/3 translate (0,2/3,0)
    A* scale 1/3 translate (0,2/3,0)
end
```

where the scale is introduced to keep the results the same size. This scene graph is equivalent to the one shown in Figure 11.8(c) and yields the limit set shown in Figure 11.9(c).

A cyclic scene graph describes the often (but not always) fractal shape that is the limit case of the L-system. The scene graph now has cycles and consists entirely of instances (no primitives), which means the resulting shape is described entirely by transformations, without any base geometry. Such structures are called (*recurrent*) *iterated function systems,* and special rendering techniques need to be used to display them (Hart and DeFanti 1991).

Figure 11.10 shows another example of the structure of scene graphs. The tops of each of the scene graphs all consist of the same union (list) node of three elements. These elements are instance nodes whose transformations are indicated by

- A—scale by 1/2 and translate (−0.433,−0.25,0)
- B—scale by 1/2 and translate (0,0.5,0)
- C—scale by 1/2 and translate (−0.433,−0.25,0)

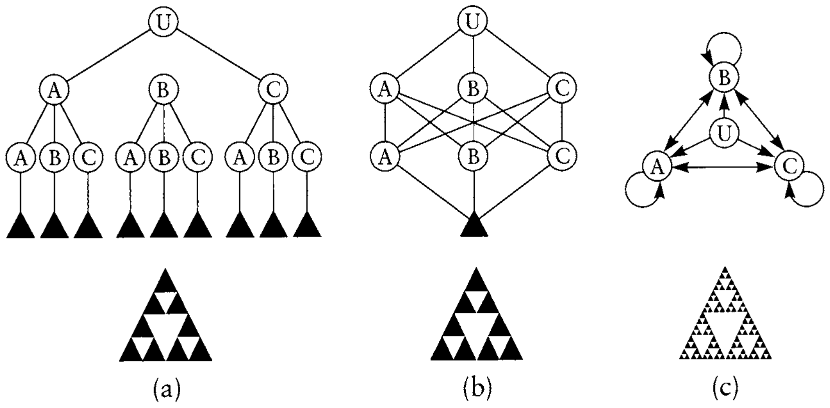

FIGURE 11.10 (a) Tree-structured scene graph, (b) directed acyclic scene graph, and (c) cyclic scene graph. The resulting scene is shown below each scene graph.

The scene graph in Figure 11.10(b) compactly combines the common subtrees of the scene graph in Figure 11.10(a). The scene graph in Figure 11.10(b) contains two copies of each node, which could be further compacted into the scene graph in Figure 11.10(c) if cycles are allowed. However, adding cycles results in a (recurrent) iterated function system, which describes objects entirely by transformations with no explicit geometric primitives (e.g., the triangle primitives are missing from the scene graph in Figure 11.10(c)).

We would like to get the power of cycles shown in Figure 11.10(c), but without the loss of geometric primitives or the difficulty in detecting cycles. We would instead like to permit conditional cycles that could terminate under some resolution-specific criterion.

The scene graph representation also provides a clear hierarchical organization of the procedural geometry for more efficient processing and rendering using lazy evaluation. A bounding volume can be stored at each of the instance nodes, and this bounding volume can be used to determine if a node's underlying geometry need be generated. For example, the earlier iterates of Figure 11.9 can serve as bounding volumes for the later iterates.

The scene graph is only capable of representing a small subfamily of L-systems, specifically deterministic nonparametric context-free L-systems without global effects such as tropism. In order to convert the more powerful extensions of L-systems into more efficient scene graphs, the scene graph will need to be augmented with the procedural extensions described in the next section.

PROCEDURAL GEOMETRIC INSTANCING

Procedural geometric instancing (PGI) augments the instance in a scene graph with a procedure, as shown in Figure 11.11. This procedure is executed at the time of instantiation (i.e., every time the object appears in a scene). The procedure has access to the instance node that called it and can change the node's transformation and shading parameters. The procedure can also change the instance to refer to a different object node in the scene graph. In addition, the procedure also has access to external global scene graph variables, such as the current object-to-world coordinate transformation matrix and passing parameters between nodes.

Parameter Passing

One enhancement to the L-system model allows it to describe shapes using real values (Prusinkiewicz and Lindenmayer 1990). Among other abilities, parametric L-systems can create more complex relationships between parent and child geometries.

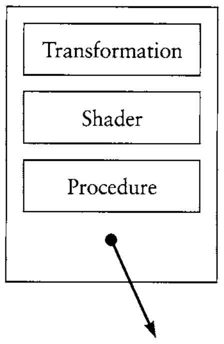

FIGURE 11.11 A procedural instance. The procedure is executed at the time of instantiation.

Parameters are by no means new to the concept of instancing. For example, they are present in SPHIGS (Foley et al. 1990), but their use here to control hierarchical subdivision differentiates them from previous implementations. Parameters are bound at instantiation in the standard fashion using a stack to allow recursion. Parameters may alter transformations or may be passed along to further instances. Both uses are demonstrated in the following example.

Example: *Inductive Instancing*

Iterative instancing builds up vast databases hierarchically, in O(log n) steps, by instancing lists of lists [Snyder and Barr 1987; Hart 1992]. For example, a list of four instances of a blade of grass makes a `clump`, four `clumps` make a `patch`, and four `patches` make a `plot`—of 64 blades of grass. Though each list consists of four instances of the same element, each instance's transformation component differs to describe four distinct copies of the geometry.

Even the step-saving process of iterative instancing becomes pedantic when dealing with billions of similar objects. Inductive instancing uses instancing parameters to reduce the order of explicit instancing steps in the definition from O(log n) to O(1). Using the field-of-grass example, we define an object `grass(n)` as a list of four instances of `grass(n-1)`, and end the induction with the definition of `grass(0)`, which instances a single blade of grass.

```
define grass(0) blade end

define grass(n)
    grass(n-1)
    grass(n-1) translate 2^n*(0.1,0.0,0.0)
    grass(n-1) translate 2^n*(0.0,0.0,0.1)
    grass(n-1) translate 2^n*(0.1,0.0,0.1)
end

grass(15)
```

Hence, a single instance of `grass(i)` inductively produces the basis of the scene in Figure 11.12 containing 4^i blades of grass.

Inductive instancing is similar in appearance to the predicate logic of Prolog. Organized properly, the defined names may be compared against the calling instance name until a match is found. For example, `grass(15)` would not match `grass(0)` but would match `grass(`*n*`)`.

Accessing World Coordinates

Objects are defined in a local coordinate frame, but are instanced into a world coordinate system. In some situations, an instance may need to change its geometry based on its global location and orientation. Given an object definition and a specific instantiation, let W denote the 4×4 homogeneous transformation matrix that maps the object to its instantiation. The transformation W maps local coordinates to world coordinates.

Procedural geometric instancing adopts the convention that within the scope of an instance, the object-to-world transformation that is available to the procedure is the one from the beginning of the instantiation and is unaffected by the instance's transformations. This solves an ordering problem where a scale followed by a

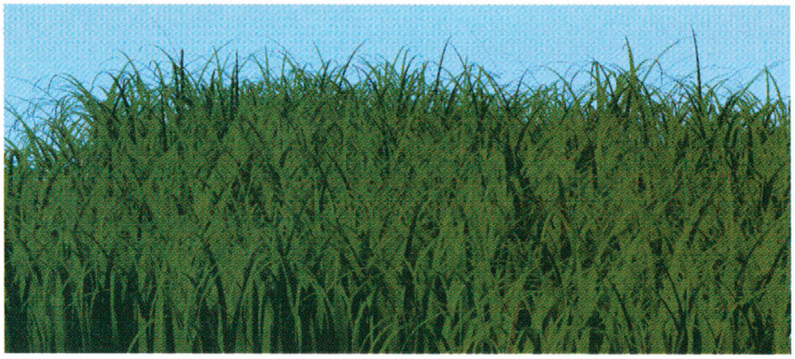

FIGURE 11.12 Grass modeled through iterative instancing, with randomness added using techniques described later in this section.

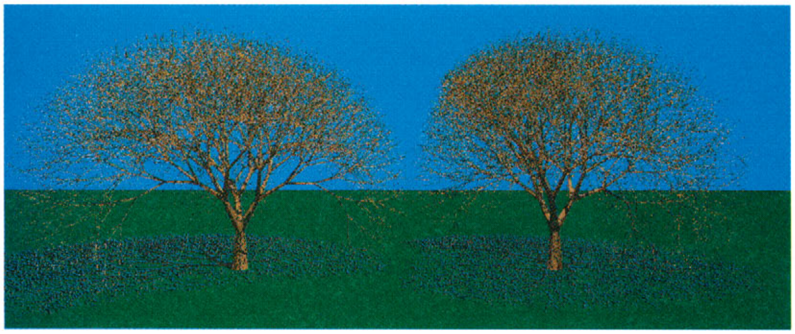

FIGURE 11.13 These trees have equivalent instancing structures except that the one on the right is influenced by downward tropism, simulating the effect of gravity on its growth.

rotation (which ordinarily are mutually commutative) might not be equivalent to a rotation followed by a scale if either depends on global position or orientation.

The following three examples demonstrate procedural models requiring access to world coordinates.

Example: *Tropism*

L-systems simulate biological systems more realistically through the use of global effects. One such effect is *tropism*—an external directional influence on the branching patterns of trees (Prusinkiewicz and Lindenmayer 1990). Downward tropism simulates gravity, resulting in sagging branches; sideways tropism results in a wind-blown tree; and upward tropism simulates branches growing toward sunlight. In each case, the tropism direction is uniform, regardless of the local coordinate system of the branch it affects.

Given its global orientation, an instance affected by tropism can react by rotating itself into the direction of the tropism. For example, the following PGI specification models Figure 11.13, illustrating two perfectly ternary trees, although one is made more realistic through the use of tropism.

```
define limb(r, l) cone(l,r,0.577*r) end
define tree(0, r, l, t, alpha)
    limb(r,l)
    leaf translate (0,l,0)
end
```

```
define branch(n, r, l, t, alpha)
    tree(n,r,l,t,alpha) rotate 30,(0,0,1)
end

define tree(n, r, l, t, alpha)
    limb(r,l)
    branch(n-1,0.577*r,0.9*l,t,alpha)
        tropism((0,1,0,0),-alpha,t)
        translate (0,l,0)
    branch(n-1,0.577*r,0.9*l,t,alpha)
        tropism((0,1,0,0),-alpha,t)
        rotate 120,(0,1,0) translate (0,l,0)
    branch(n-1,0.577*r,0.9*l,t,alpha)
        tropism((0,1,0,0),-alpha,t)
        rotate 240,(0,1,0) translate (0,l,0)
end

tree(8,.2,1,(0,-1,0),0) translate(-4,0,0)
tree(8,.2,1,(0,-1,0),20) translate(4,0,0)
```

The procedure `tropism` is defined as

$$\texttt{tropism}(\mathbf{v}, \alpha, \mathbf{t}) \equiv \texttt{rotate}\ \alpha\ ||W\mathbf{v} \times \mathbf{t}||, W\mathbf{v} \times \mathbf{t} \tag{11.6}$$

This `tropism` definition may be substituted by hand, translated via a macro, or hard-coded as a new transformation.

The object `limb` consists of an instance of `tree` rotated 30°. Under standard instancing, this object could be eliminated, and the 30° could be inserted in the definition of `tree`. However, this operation is pulled out of the definition of `tree` because the world transformation matrix W used in the definition of `tropism` is only updated at the time of instantiation. The separate definition causes the tropism effect to operate on a branch after it has rotated 30° away from its parent's major axis.

Tropism is typically constant, although a more accurate model would increase its severity as branches become slimmer. Thompson (1942) demonstrates that surface tension dictates many of the forms found in nature. In the case of trees, the strength of a limb is proportionate to its surface area, $l \times r$, whereas its mass (disregarding its child limbs) is proportionate to its volume, $l \times r^2$. We can simulate this by simply increasing the degree of tropism α inversely with respect to the branch radius r. Hence, `branch` would be instantiated with the parameters

```
branch(n-1,0.577*r,0.9*l,t,(1-r)*alpha0)
```

where `alpha0` is a constant, maximum angle of tropism.

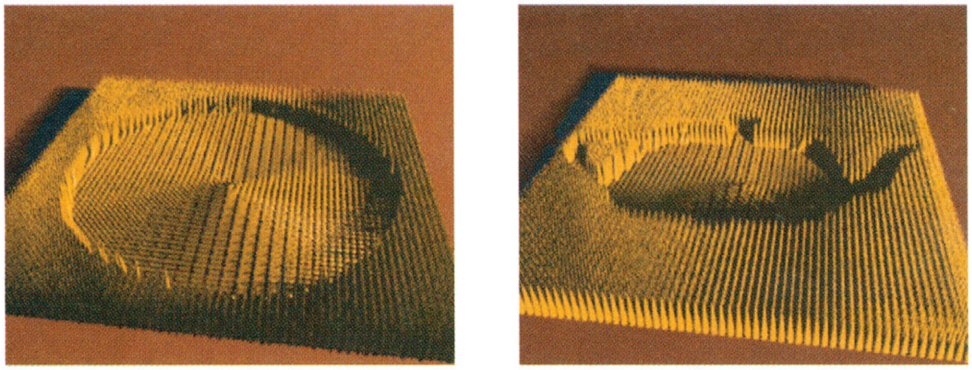

FIGURE 11.14 Crop circles trampled in a field of cones.

While variable tropism could be incorporated, via source code modification, as a new parameterized symbol in the turtle-based L-system paradigm, procedural geometric instancing provides the tools to articulate it in the scene description.

Example: *Crop Circles*

Prusinkiewicz, James, and Mech (1994) added a query command "?" to the turtle's vocabulary that returned the turtle's world position. This information was used to prune L-system development based on external influences. The world coordinate transformation can similarly detect the presence of an external influence, and the geometry can appropriately respond. A crop circle such as those allegedly left by UFO encounters is similar to the simulation of topiary, except that the response of pruning is replaced with bending as demonstrated in Figure 11.14.

Such designs can be described implicitly (Prusinkiewicz, James, and Mech 1994) in the case of a circle, or through a texture map (*crop* map) (Reeves and Blau 1985) in the case of the teapot.

Example: *Geometry Mapping*

The grass example from the previous section shows how geometry can be instanced an arbitrary number of times onto a given section of a plane. Consider the definition of a Bezier patch as a mapping from the parameter plane into space. This mapping takes blades of grass from the plane onto the Bezier patch. Replacing the blades of grass with fine filaments of hair yields a fully geometric specification of fur or hair.

Other Functions

In addition to passing parameters and accessing world coordinates, several other features based on these abilities make specification of procedural models easier.

Random Numbers

Randomness can simulate the chaos found in nature and is found in almost all procedural natural modeling systems. Moreover, various kinds of random numbers are useful for natural modeling.

The notation `[a,b]` returns a random number uniformly distributed between `a` and `b`. The notation `{a,b}` likewise returns a Gaussian-distributed random number.

The Perlin *noise* function provides a band-limited random variable (Perlin 1985) and is implemented as the scalar-valued function `noise`. A typical invocation of the noise function using the world coordinate position is specified: `noise(W(0,0,0,1))`.

Example: *Meadows*

Fractional Brownian motion models a variety of terrain. This example uses three octaves of a $1/f^2$ power distribution to model the terrain of a hilly meadow. Grass is instanced on the meadow through a translation procedurally modified by the noise function. The placement of the grass is further perturbed by a uniformly random lateral translation, and its orientation is perturbed by the noise function.

The following PGI scene specification describes the meadow displayed in Figure 11.15. The vector-valued function `rotate(x,theta,axis)` returns the vector `x` rotated by `theta` about the axis `axis`. Its use in the definition of `fnoise` disguises the creases and nonisotropic artifacts of the noise function due to a simplified interpolation function.

```
#define NS 16 /* noise scale */
#define fnoise(x) (NS*(noise((1/NS)*(x)) +
                  0.25 noise((2/NS) rot((x),30,(0,1,0))) +
                  0.0625 noise((4/NS) rot((x),60,(0,1,0)))))
#define RES 0.1 /* polygonization resolution */
```

```
define plate(-1)
    polygon (-RES,fnoise(W(-RES,0,-RES,1)),-RES),
            (-RES,fnoise(W(-RES,0, RES,1)), RES),
            ( RES,fnoise(W( RES,0, RES,1)), RES)
    polygon ( RES,fnoise(W( RES,0, RES,1)), RES),
            ( RES,fnoise(W( RES,0,-RES,1)),-RES),
            (-RES,fnoise(W(-RES,0,-RES,1)),-RES)
end

define plate(n)
    plate(n-1) translate 2^n*( RES,0, RES)
    plate(n-1) translate 2^n*(-RES,0, RES)
    plate(n-1) translate 2^n*( RES,0,-RES)
    plate(n-1) translate 2^n*(-RES,0,-RES)
end

define blade(0) polygon (-.05,0,0),(.05,0,0),(0,.3,0) end

define blade(n)
    blade(n-1)
      scale (.9,.9,.9) rotate 10,(1,0,0) translate (0,.2,0)
    polygon (-.05,0,0),(.05,0,0),(.045,.2,0),(-.045,.2,0)
end

define grass(-2)
    blade(10)
      rotate 360*noise((1/16)*(W(0,0,0,1))),(0,1,0)
      translate [-.05,.05],fnoise(W(0,0,0,1)),[-.05,.05])
end
```

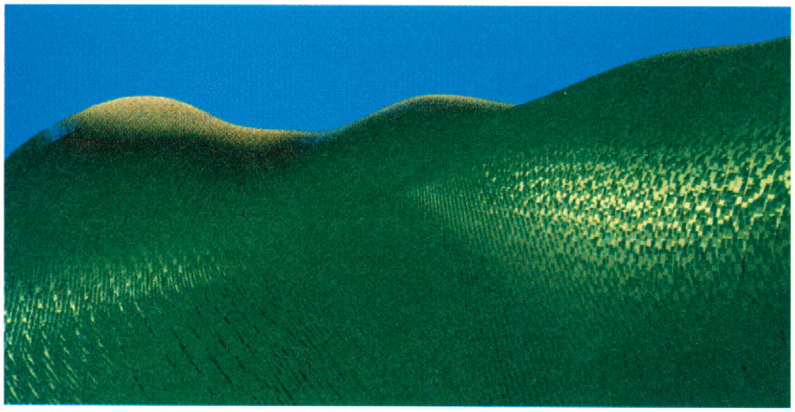

FIGURE 11.15 A grassy meadow with an exaggerated specular component to demonstrate reflection "hot spots." No texture maps were used in this image.

```
define grass(n)
    grass(n-1) translate 2^n*(.1,0,.1)
    grass(n-1) translate 2^n*(-.1,0,.1)
    grass(n-1) translate 2^n*(.1,0,-.1)
    grass(n-1) translate 2^n*(-.1,0,-.1)
end

plate(6)
grass(6)
```

Levels of Detail

The scale at which geometry projects to the area of a pixel on the screen under the rules of perspective is bound from above by the function

$$\texttt{lod}(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_0||\ 2\tan(\theta/2)/n \tag{11.7}$$

where $\mathbf{x}_0$ is the eyepoint, θ is the field of view, and n is the linear resolution. The condition `lod`$(W(0,0,0,1)) > 1$ was used to halt recursive subdivision of fractal shapes constructed by scaled instances of the unit sphere (Hart and DeFanti 1991). In typical use, `lod` replaces complex geometries with simpler ones to optimize display of detailed scenes.

Comparison with L-Systems

L-systems are organized into families based on their representational power. The simplest family is the deterministic context-free L-system. Parameters were added to handle geometric situations requiring nonintegral lengths (Prusinkiewicz and Lindenmayer 1990). Stochasticism was added to simulate the chaotic influences of nature. Various degrees of context sensitivity can be used to simulate the transmission of messages from one section of the L-system model to another during development. Global influences affect only the resulting geometry, such as tropism, which can simulate the effect of gravitational pull when determining the branching direction of a tree limb.

Figure 11.16 depicts the representational power of procedural geometric instancing with respect to the family of L-system representations. Standard geometric instancing can efficiently represent only the simplest L-system subfamily, whereas there is currently no geometrically efficient representation for any form of context-sensitive L-system. Procedural geometric instancing is a compromise, efficiently representing the output of a stochastic context-free parametric L-system with global effects.

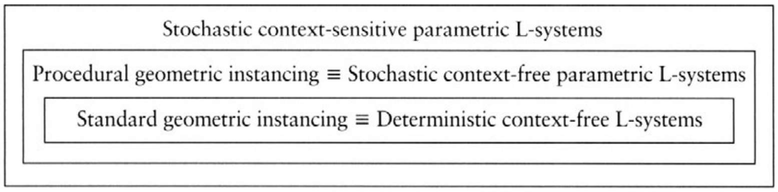
Stochastic context-sensitive parametric L-systems
Procedural geometric instancing ≡ Stochastic context-free parametric L-systems
Standard geometric instancing ≡ Deterministic context-free L-systems

FIGURE 11.16 Hierarchy of representations.

Ordering

Several rendering methods benefit from a front-to-back ordering of the geometry sent to it. We order our bounding volume hierarchy through the use of axially sorted lists. The contents of each list are sorted six times, in nonincreasing order of their most extreme points in the positive and negative x, y, and z directions. Each list is then threaded six times according to these orderings. At instantiation, we determine which of the six vectors, W^{-1T} $(\pm1,0,0,0)$, W^{-1T} $(0,\pm1,0,0)$, or W^{-1T} $(0,0,\pm1,0)$, has the maximum dot product with a unit vector from the position W $(0,0,0,1)$ pointing toward the viewer. The elements of the list are then instantiated according to this order. While not perfect, this method does an effective job of instancing geometry globally in a front-to-back order.

BOUNDING VOLUMES

The bounding volume of procedural geometry must account for the different possible shapes the procedural geometry may take. A bounding volume is associated with every node in a PGI scene graph, and the procedure associated with the node can alter the bounding volume of the instantiation.

Early terrain models simulated fractional Brownian motion by midpoint displacement, which was an early subdivision surface method for triangle or quadrilateral meshes (Fournier, Fussel, and Carpenter 1982). Later systems computed the maximum possible extent of the midpoint displacement surface of a mesh element and used this extent to construct a bounding volume for the mesh element (Kajiya 1983; Bouville 1985). This supported the lazy evaluation of the procedural terrain model. If the renderer determines that the bounding volume of a mesh element is not visible, then that element need not be subdivided and displaced.

Bounding volumes for procedural geometry can be computed statically or dynamically. A *static* bounding volume encases all possible geometries synthesized by the procedure, whereas a *dynamic* bounding volume is designed to tightly bound the specific output of a particular evaluation of the procedure.

Dynamic bounding volumes are more efficient than static bounding volumes, but they are also much more difficult to devise. Dynamic bounding volumes need to be computed at the time of instantiation. A bounding volume procedure is executed that predicts the extent of the procedural geometry based on the parameters used to synthesize the geometry.

Example: *Bounding Grass*

The bounding box hierarchy for the field of grass specified previously in an example is constructed by the dynamic bounding volume

$$\texttt{bounds}((0,0,0),2^{n+1}(.1,0,.1) + (0,0.3,0))$$

where `bounds` specifies a bounding box with two of its opposing corners. Each blade of grass is assumed to be 0.1 units thick and 0.3 units high.

Example: *Bounding Trees*

Under standard geometric instancing, a tree may be specified as a trunk and several smaller instances of the tree. Repeated forever, this generates a linear fractal, and the instancing structure is similar to an iterated function system (Hart 1992). Given an iterated function system, the problem of finding an optimal bounding volume for its attractor remains open, although several working solutions have appeared (Hart and DeFanti 1991; Canright 1994; Dubuc and Hamzaoui 1994).

Under procedural geometric instancing, tree subdivision gains additional freedom for global influences. Beginning from the trunk, it is a matter of chaos where each leaf is finally instanced. Bounding volumes are derived from prediction, which is difficult in such chaotic cases. Hence, we look to the worst possible case to obtain an efficient parametric bound on the geometry of a tree.

Trees, such as those described previously in an example, scale geometrically, with each branch a smaller replica of its parent. Geometric series, such as the length of branches from trunk to leaf, sum according to the formulas (and PGI built-in functions)

$$\sum_{i=0}^{\infty} x^i = \mathtt{sumx}(x) = \frac{1}{1-x} \tag{11.8}$$

$$\sum_{i=m}^{n} x^i = sumxmn(x, m, n) = \frac{x^m - x^n}{1-x} \tag{11.9}$$

Consider the definition of a tree whose branches scale by factors of λ_i. Let $\lambda = \max \lambda_i$ be the scaling factor of the major branch; for sympodial trees (Prusinkiewicz and Lindenmayer 1990), this is the extension of the trunk. Then from the base of the trunk, which is canonically set to unit length, the branches can extend no farther than $\mathtt{sumx}(\lambda)$ units away. When limited to n iterations, the branches are bound by a sphere of radius $\mathtt{sumxmn}(\lambda, 0, n)$ centered at the base of the trunk. At iteration m, these branches are bound by a sphere of radius $\mathtt{sumxmn}(\lambda, m, n)$ centered at the current point. While these bounds are quite loose initially, they converge to tighter bounds at higher levels.

While tight bounding volumes of near-terminal limbs will efficiently cull the rendering of hidden branches, a tight bound on the entire tree is also useful to avoid rendering trees that are hidden behind dense forests or mountain ridges. A simple worst-case analysis can precompute the maximum height a tree attains as well as the maximum width. These computations are less useful for the branches because the global influences may be different on the arbitrarily positioned and oriented instance of the tree. However, clipping the bounding volumes of the branches to the tree's bounding volume makes the branch bounding volumes tighter and more efficient.

CONCLUSION

Procedural geometric instancing is a language for articulating geometric detail. Its main impact is the insertion of procedural hooks into the scene graph. These hooks allow the scene graph to serve as a model for procedural geometry and allow it to overcome the intermediate storage problem. The resulting scene graph provides on-demand lazy evaluation of its instances performed at the time of instantiation, and only for objects affecting the current rendering of the scene.

Procedural geometric instancing is a geometric complement to shading languages. It provides the renderer with a procedural interface to the model's geometry definition in the same way that a shading language provides the renderer with a procedural interface to the model's shading definition.

Procedural geometric instancing yields geometries that are processed more efficiently than those generated by turtle graphics. Procedural geometric instancing is also based on the scene graph and its associated scene description, which is a more familiar and readable format for the articulation of procedural models than is an L-system's productions of turtle graphics symbols.

The parameterization and other features of procedural geometric instancing make standard textual geometric descriptions of natural models more compact and readable.

Procedural Geometric Modeling and the Web

Procedural modeling can play a critical role in multimedia, networking, and the World Wide Web. Two standards have recently become popular: Java and VRML. Java is a system that allows programs to be automatically loaded from a remote site and run safely on any architecture and operating system. The Virtual Reality Modeling Language (VRML) has become a standard for transmitting geometric scene databases.

In their current form, VRML geometric databases are transmitted in their entirety over the network, and Java has little support for generating complicated geometric databases. However, both can be enhanced to support the lazy evaluation paradigm for procedural modeling.

One example extends the capabilities of Java or an equivalent language to support the generation and hierarchical organization of detailed geometry. A user may download this script, and a renderer then runs it to procedurally generate the necessary geometry to accommodate the vantage point of the viewer. This example places the network at the "articulation" step of the paradigm.

A second example places the network at the "geometry/coordinates" bidirectional step of the lazy evaluation paradigm. In this example, a powerful server generates the geometry needed by a remote client renderer to view a scene and transmits only this geometry over the network. As the client changes viewpoint, the server then generates and transmits only the new geometry needed for the new scene.

Future Work

The major obstacle to efficient rendering for procedural geometric instancing is the construction of effective bounding volume hierarchies. Since geometry is created on demand, a bounding volume must be able to predict the extent of its contents. Such procedural bounding volumes were constructed for fractal terrain models by Kajiya

(1983) and Bouville (1985), but their generalization to arbitrary subdivision processes remains unsolved.

Amburn, Grant, and Whitted (1986) developed a system in which context was weighted between independent subdivision-based models. Fowler, Prusinkiewicz, and Battjes (1992) developed a geometric context-sensitive model of phyllotaxis based on the currently generated geometry of a procedural model. A similar technique could model the upward tropism of the tips of branches on some evergreen trees. These tips bend upward depending on the visibility of sunlight. If the tree was instanced from the highest branches, working its way down to the ground level, the visibility of each branch with respect to the previously instanced branches could be efficently computed using a separate frame buffer as a progressively updated "light" map.

ACKNOWLEDGMENTS

Procedural geometric instancing has been in various forms of development for nearly a decade, and some of this development was funded by Intel. Anand Ramagapalrao implemented the front-to-back sorting of hierarchical bounding volumes. Chanikarn Kulratanayan and Hui Fang also used the system for their research and generated numerous impressive example images. A significant portion of the ideas behind PGI resulted from conversations with Przemyslaw Prusinkiewicz, most occurring on various ski lifts across the Northwest.