GENETIC TEXTURES

F. KENTON MUSGRAVE

INTRODUCTION: THE PROBLEM OF PARAMETER PROLIFERATION

As we saw in Chapter 15, one problem confronting us in the construction of procedural textures or shaders is that of *parameter proliferation*. The "terran" texture presented in Chapter 15 has 22 user-definable parameters, plus some 51 hard-coded constants hidden in the shader code. The results of changes to these parameters are often far from obvious and sometimes downright bizarre. For use in a production environment by artists who didn't write the shader, this kind of situation is simply absurd. Most users of shaders—digital artists—don't know about, or want to be confronted with, the rich logical complexity and odd machinations of the shader's internal operation. They generally have another job to get done: making pictures, within rigid constraints on time and quality. It is unfair and counterproductive to require them to learn about or necessarily understand how shaders work. And yet the terran texture illustrates how, in fact, shaders are built and operate.

On the other hand, neither is it all that easy for the programmer or shader writer to define and implement all those parameters. It's tedious, arduous, and generally time consuming to write and refine such complex shader code. The ultimate product that the programmer should ideally deliver is a simple, intuitive user interface that's easy for an artist, with no programming or math background, to understand and use. Unfortunately, if effective shader writing is a black art, then devising such interfaces is a black hole.

I hate to sound defeatist here, but ever since Gavin Miller first pointed out this problem to me in 1988, when he was working at what is now Alias/Wavefront, I have been bemused by this problem. I have not met anyone who has an effective, general strategy for reducing a huge number of parameters to a few intuitively obvious sliders that maintain the power of the underlying functionality. I obtained direct experience in managing this problem when producing the Disney Cruise Lines commercial, where Mickey and Goofy are made out of clouds, at Digital Domain in the spring of 1997. In that case, we—the art director and myself—simply revealed

parameters to the artists one at a time, as we perceived that they needed them or that the look would benefit from their knowing about them. The vast majority of parameter values were preset by myself, the programmer. Unfortunately, this requires that the programmer also be something of an artist, something that is not always possible to achieve in a production context.

A USEFUL MODEL: AESTHETIC *n*-SPACES

Here is one way of thinking about the textures and their controls that I find useful: they represent an *aesthetic n-space*. The "aesthetic" part simply means that changing values of the parameters affects the aesthetics of the result. The "*n*-space" part is more subtle. The "*n*" in "*n*-space" is simply some whole, positive number, from zero to infinity. Each separate one of those *n* numbers represents a *degree of freedom*. Think of a degree of freedom as a new direction in which we can move. For $n = 1$ we have a line and exactly one axis along which we may move in two directions, call them left and right, for convenience. For $n = 2$ we have a plane, wherein we may move left and right and, say, forward and backward. For $n = 3$, we have the familiar three-space in which we live, wherein we may move left and right, forward and backward, and up and down. When *n* equals 4 or higher, we move into the higher dimensions for which human intuition fails us but into which mathematicians never hesitate to go.

In this model, the terran texture presented in Chapter 15 represents a 73-dimensional aesthetic space! No wonder you wouldn't want to hand that shader over, as written, to a digital artist. Believe me, it took more than a few hours to define and determine values for those 73 parameters, too. How can we deal with these two problems, the overwhelmed user and the overworked programmer? *Genetic programming* can provide a fascinating solution to both. But before I describe exactly what genetic programming is, let me first describe some motivating concepts behind our process.

The process of defining the *n* parameters in a procedural texture corresponds to the *creation* or *specification* of the *n*-space. The process of determining good values for the parameters may be thought of as *searching the n-space for local maxima of an aesthetic gradient function.*[1] This is an abstraction of which I am particularly fond: as we change the values of the parameters, we move about in the *n*-space, in a manner exactly analogous to the low-dimensional spaces described earlier. As we move

1. This model is based on what are called hill-climbing optimization methods, such as simulated annealing (Press et al. 1986).

about, the aesthetics change. How they change is determined by an entirely subjective aesthetic judgment on the part of the user. But clearly some sets of values will provide images that are "better" and other images that are "worse." The aesthetic gradient function is then the user's subjective evaluation of how the "goodness" of the result changes with changes in the parameter values, the gradient being between "better" and "worse." A local maximum represents a set of parameter values where, if any one is changed a little, the image gets worse. Thus we're in a position analogous to being on a local hilltop or local aesthetic maximum. Small movements in all directions in n-space correspond to moving downhill in terms of our aesthetics. Yet this hilltop is only local—there is no guarantee that, if we move far enough away from our current point, we'll cross the equivalent of some "aesthetic valley" and be able to climb up a higher hill to a better local aesthetic maximum. The nice thing about this model of the creation and search of n-space is that it is independent of the value of n and therefore of the complexity of the texture or shader.

CONTROL VERSUS AUTOMATICITY

An inevitable outcome of the growth of complexity (e.g., the number of parameters) is that there arises an eternal tension that is general to models for image synthesis: control versus ease of use. If you make things clear, simple, and easy for the user, you necessarily have to compromise control, because control lies in the complexity of the procedures. If you give the user full control, the interface becomes overwhelming in its baroque complexity. Anyone who's used 3D modeling or rendering software, from low-end consumer to the top-of-the-line professional packages, has confronted this problem.

As we've tried to make clear through much of this book, the whole paradigm of proceduralism is intimately caught up with this idea of *amplification,* described in Chapter 14, whereby lots of visual detail issues from a relatively small number of controls (parameters). Unfortunately, the flip side of this wonderful power is that automaticity implies lack of control. Just as in any human project large enough to require delegation of subtasks to colleagues, you abdicate full control over the results. Thus we may construct the beautiful planet Gaea, imbued with the capacity to be imaged at any range and/or field of view and resolution (see Figures 16.3 and 16.6), from a very small amount of computer code, but we cannot, without compromising elegance, control any of the specific features found there. We have only qualitative controls with global effects.

Nevertheless, we can obtain some striking and useful results. But what if we take this amplification/automaticity to its logical extreme and let the computer do

everything? Then the user would simply sit back and pick and choose from various offerings, like Scarlet O'Hara selecting a beau for a dance. In this paradigm, the computer simultaneously specifies and searches the aesthetic *n*-space. The method is spectacularly productive, wonderfully automatic—once a lot of programming has been done—but difficult to direct to a desired end, for example, a wood-grain texture. From a practical point of view, this last point may be a fatal flaw. Yet I'll describe the genetic programming paradigm here because it illustrates the functional nature of procedural textures and clarifies through extreme abstraction how such textures are built, and simply because genetic programs are the most fun software systems I've ever played with. Because the details of implementation are tedious, but the concepts driving them are quite clear, I'll stick to a high-level description here. The code for the genetic program called Dr. Mutatis that I wrote for MetaCreations went the way of that ill-fated company and is no longer available, unfortunately.

One thing you might keep in mind when reading this chapter is my basic motivation for taking on the nontrivial task of programming genetic textures: breeding planets. You see, since 1987 I've been working toward building a synthetic universe. One big problem: a universe has a *lot* of planets—far too many to build by hand with code as complex as the terran texture. The solution? Have the computer build them for us, automatically—or as close to "automatically" as we can get. The synthetic universe we're out to build is meant to be full of surprise and serendipity, so exact control of all the details is not important. We're not out to build a preconceived stage set, but rather to explore some of the beauty inherent in mathematics and logic, the mathematics and logic embodied in the texture code. This gets rather philosophical and is covered in the final chapter of this book. For now let's jump into the genetic approach to building procedural textures.

A MODEL FROM BIOLOGY: GENETICS AND EVOLUTION

Genetic programming starts with a model borrowed from biology and proceeds to use it by analogy. The idea was introduced to the computer graphics community by Karl Sims in his 1991 SIGGRAPH paper (Sims 1991). Sims in turn got the idea from Richard Dawkins's book *The Blind Watchmaker* (Dawkins 1987) and the simple computer graphics program called BioMorph that Dawkins uses to illustrate the power of the theory of evolution in explaining the origins and complexity of life on Earth. Unlike Dawkins, I have no metaphysical ax to grind vis-à-vis the origin of life or competing religious and scientific models for the origin of life. I simply have found genetic programming to be the coolest thing you can do with procedural

textures and computer art and an important stepping stone toward the future of proceduralism that I envision, this synthetic universe.

We start with a few definitions that should be familiar from your high school biology classes. Recall that the *genotype* is the genetic description for a given organism. The genotype is encoded in a fantastically long molecule of *deoxyribonucleic acid* (DNA). The genotype is a specific instance from a *genome,* as in the Human Genome Project, the major scientific initiative that has mapped the general layout of all human genes. The genome is general to a species and has variations among individuals, while the genotype is specific to a single organism. A *gene* is a specific part of the genome that encodes a certain function, generally instructions for building a certain biologically active protein.

The *phenotype* is the physical manifestation of the instructions encoded in the genotype: it is a specific organism, such as you or I. Your genotype is similar to mine, but they are not identical, so while we are both human beings, we are not identical twins or clones. Different instances of a given genotype will, given similar environments during development, reliably give rise to a certain, well-defined phenotype, as with identical twins (who are, in fact, clones).

Charles Darwin's famous and controversial theory of evolution posits that life has literally risen from the primordial ooze through *evolution*—the refinement of genomes through what he called *natural selection:* the preferred survival and propagation of individuals whose genotype has given rise to a "more fit" phenotype.[2]

Natural selection acts on phenotypes. No progress would occur if the phenotypes didn't change over generations. In nature they do, by two mechanisms: *mutation* and *sexual reproduction.* Mutation occurs through errors introduced in the DNA replication process and by direct alterations to DNA molecules by external mechanisms such as ionizing radiation (for example, ultraviolet light and radioactive decay byproducts). Sexual reproduction is presumed, in biology, to be a clever adaptation by higher organisms. In sexual reproduction, genes from two parents are mixed and matched to "reshuffle the deck," providing random combinations of proven genes. This is a safer strategy for productive change than the purely random variations provided by mutation, most of which presumably will not produce viable phenotypes.

---

2. Note that Darwin's theory of evolution preceded the discovery of DNA by about a century. The theory of evolution is predicated only on the idea of *inheritance,* whereby offspring acquire the genetic information of their ancestors. It is in no way dependent on the mechanism for encoding or passing on that information. This independence helps bolster the analogy we're making here.

Evolution is the accumulation of "improvements" in the phenotypes through these changes, as culled by natural selection, in inheritable genomes. This process can be convincingly argued to account for all the glorious complexity and variation in life on Earth, as Dawkins's series of books on the topic attempts to do.

I have come to think of DNA as being like the operating system (OS) of an organism, while the cells of which the organism is composed are like the computer on which the OS runs. (Fortunately, the human OS is generally more reliable than those we've devised for our computers.) The coding of both DNA and an OS is very abstract: the program code that comprises a computer OS bears little resemblance to the user interface it presents to us, just as the DNA molecule little resembles a hamster, a redwood tree, or you. Also similarly, both are highly nonportable: the encoding is practically worthless without the platform on which it is designed to execute. Hence you can't just install and run the Mac OS on a PC, nor can you put human DNA into a starfish cell nucleus and expect to grow a healthy human baby. This is a little unfortunate for our application here, as we'd like to have a universal computer genome that could be run on any computer, so that we could develop a kind of universal artificial life, or "A-life" as it's called.

The Analogy: Genetic Programming

What we're interested in here is procedural textures and how to create beautiful ones efficiently. Enter our analogy: We will regard the code that specifies a texture to be its genotype and an image of the resulting texture to be the corresponding phenotype. Evolution is directed by what I call *unnatural selection:*[3] God-like intervention by the user, deciding which phenotypes, and thus the underlying genotypes, survive and propagate. Change in the genome is accomplished by methods analogous to mutation and sexual reproduction: we design the program to introduce random variations in genotypes and to be able to share "genes" in an analog of sexual reproduction.

What then, is a gene in a genetic texture program? As with DNA, it is a unit of genetic "code" specifying some functionality within the resulting texture, for example, an fBm procedure. DNA is composed of the four nucleic acids cytosine, guanine, adenosine, and thymine, commonly referred to by their initials C, G, A, and T. The

3. I like this tongue-in-cheek term "unnatural selection" because it points out the artificial separation of humankind from nature. In my view, humans and their actions are natural phenomena. If you disagree, I suggest you try—in a thought experiment—separating humans entirely from nature and see how we do!

DNA molecule is a long sequence of these *bases,* as they are called, paired across from one another in the famous double helix. A certain functional sequence of bases can comprise a gene.

Our encoding scheme is a little different. Our bases are all complete functions, analogous in DNA more to genes than to bases. A combination of our bases can, however, function as a gene. The difference between genes and bases is that bases are *atomic:* bases cannot be subdivided into smaller parts.

The encoding scheme for our genetic information is, rather than a linear sequence as in DNA, an *expression tree,* which is analogous to a genealogical family tree. (See Figure 19.1.) A tree is a special kind of graph. The graph is composed of *nodes.* There are two relevant kinds of relationships between nodes: *parent* and *child,* the meaning of which is obvious. There are three types of nodes: the *root* node at the top of the tree (although it might seem that it should be called the bottom), which has no parent and usually has children; *interior* nodes, which have both parents and children; and *leaf* nodes, which have parents but no children. The root node generally has to return three values to create an RGB value to display as the phenotype.

The expression tree operates via *functional composition,* described in Chapters 2 and 15 as *perturbation* or *domain distortion.* The idea of functional composition is simply that a function takes as its input parameters the output of another function or functions. We saw the effects of simple functional composition in those earlier chapters; now we take the idea to an extreme. In the expression tree, only leaf nodes

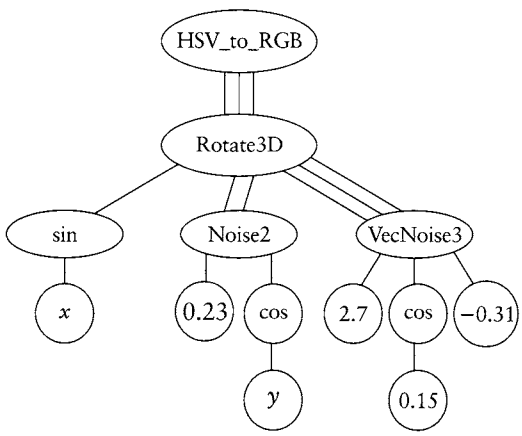


FIGURE 19.1 An expression tree. The circles are nodes; the lines between them are *links* representing relationships.

provide values that are not determined by functions. (In fact, the leaf nodes are usually simple linear functions of x, y, or z; that is, they are simply the x, y, or z value of the point where the texture is being evaluated. The rest are simply random numbers.)

Expression trees "evolve" via random mutation and sexual reproduction, as illustrated in Figures 19.2 and 19.3. The user selects which phenotypes mutate and breed.

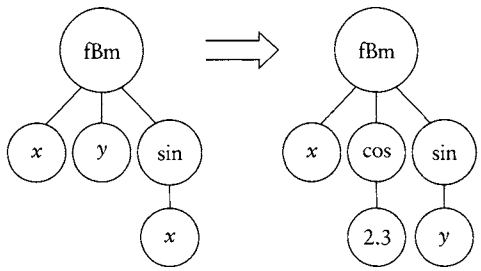


FIGURE 19.2 Mutation in an expression tree.

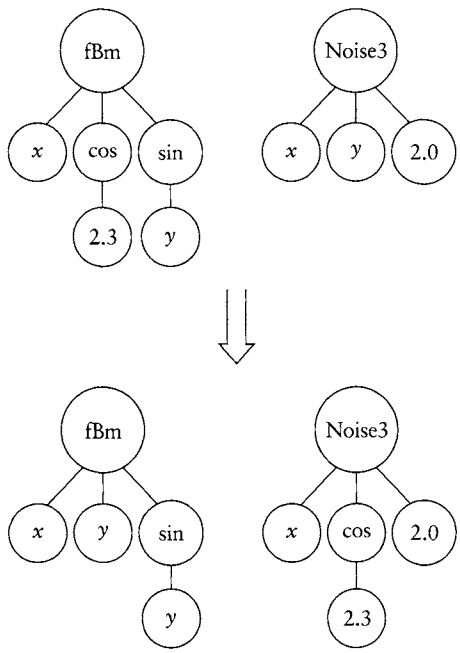


FIGURE 19.3 Sexual reproduction between two expression trees.

Implementation

The expression tree is perhaps most easily implemented in a high-level, functional language such as LISP, as in Sims's original genetic texture program. Unfortunately, LISP is relatively slow, unless you happen to have something like a Connection Machine 2 to run it on. Indeed, my first experience with genetic textures involved running Karl's LISP code on a CM-2 using 16,000 processors. Ordinary users like you and me will require a more efficient implementation for our more pedestrian computers. In fact, I developed my genetic texture code primarily on a laptop, showing how far processors have come since 1991. Steven Rooke tells me that switching from LISP to C++ sped his genetic program by a factor of 100. (And yes, I know LISP fans who'd contest that!)

It turns out that genetic programming is a perfect application for C++. The data structures required for the various types of nodes, their relationships, and the operations defined upon them are succinctly described in C++ classes. The devil is in the details, mostly in memory management. For efficiency, I have each evaluation of a function in the tree process an entire scanline's worth of data. (This saves a whole lot of tree traversals that would be necessary if you evaluate the tree pixel by pixel.) For large trees, this can get into a lot of memory. Again for efficiency, I do all my own memory management in the program because the C and C++ memory allocation routines `malloc` and `new` are relatively slow. So I end up with piles of pointers and memory pools—and lots of room for bugs. But that's just standard programming and thus outside the scope of this book.

There are two interesting issues in programming genetic textures that I'd like to point out: the meaning of the root node, in terms of color, and the effect that a given library of genetic bases has on the kinds of images produced.

INTERPRETATION OF THE ROOT NODE

Ultimately, we want to make color images. Thus each pixel will require a separate value for red, green, and blue. A solution that immediately pops into mind is that we simply have the root node consist of three separate subtrees, one each for red, green, and blue. In practice, however, this is usually unsatisfactory: you tend to end up with unrelated, overlaid images in red, green, and blue. You can interpret the values as lying in another color space, such as HLS (hue/luminosity/saturation), but similar problems remain. The usual solution is to have the root function return a single value that serves as an index into a color lookup table. This brings up the separate and unrelated problem of generation of, and making changes to, that lookup table. It is easy enough to automatically generate random color maps (Musgrave 1991),

but the obvious solutions are a little inelegant compared to the rest of our fully procedural paradigm.

The solution I'm currently using is based on the ideas behind the random color textures presented in Chapter 15. The root node comprises a three-vector valued function, which has been passed through a random rotation matrix to correlate, in the final RGB color space, the influence of the three components of the vector. Mathematically, we'd say that the three vector components are then *linear combinations* of the *basis vectors* that correspond to red, green, and blue. That is, rather than having each component of the three-vector mapping to only red, green, or blue, each component contributes to all of red, green, and blue, albeit indirectly through a final HLS to RGB transform (see the expression tree in Figure 19.1). From a mathematical perspective this insight is obvious; my apologies if it's not exactly clear when translated into English prose. Such is the divergence of the two modes of thinking and communication. But it's cool to see again, as in the GIT schemes described in Chapter 15, that a mathematical perspective can provide useful aesthetic insights.

This approach is not without its own problems. First, constructing a random rotation matrix requires at least six input values: two—altitude and azimuth—to specify the random rotation axis, one to specify the angle of rotation, and three to specify the vector being transformed by the resulting matrix. This implies a bushy tree at the root, which in turn implies increased evaluation time. It also implies a rather large amount of storage in the vector class—a maximum of six double-precision floating-point numbers,[4] which adds up when you need to store a lot of vectors. This approach also tends to consistently produce rainbows, due to the final HLS to RGB transform. These rainbows become boring to annoying but can be excised through the process of unnatural selection. The advantage of the approach, as I see it, is that it nicely preserves the pure functional paradigm.

THE LIBRARY OF GENETIC BASES

An important aspect of any genetic image generation program is its library of bases, the functions out of which the expression trees are formed. This library literally provides the expressive vocabulary of the system. Thus different genetic texture programs have different "looks." Karl Sims's system has a library of primitive mathematical functions such as sine, log, arc tangent, and so on, as well as iterated

4. My experience indicates that single-precision floating point does not provide sufficient accuracy for the kind of multiple functional composition involved in genetic textures.

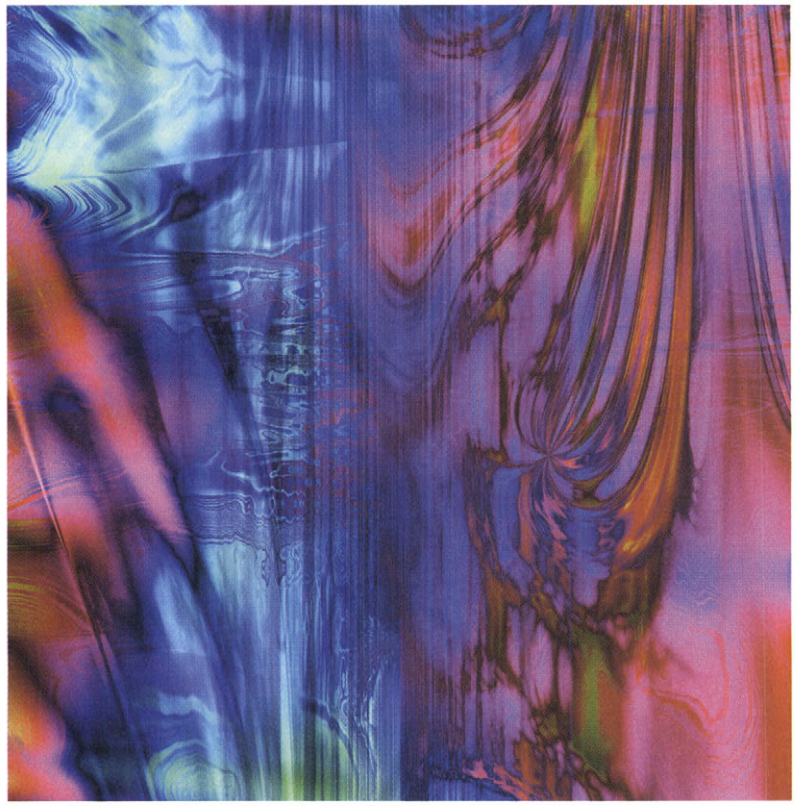
function systems (Sims 1991) that give rise to characteristic fractal patterns. Steven Rooke's system (*www.azstarnet.com/~srooke/*) is heavy on deterministic fractal functions that are generally iterations on the complex plane, including a genetic generalization of the kinds of functions people experiment with in the well-known Fractint freeware program. My own system (*www.kenmusgrave.com/mutatis.html*) is based primarily on the kind of random fractal functions described in Chapters 14 and 16. Thus each system tends to create images with a certain, fairly consistent character. Certainly, each produces images that the others are not capable of generating, due to the expressive limitations of their respective libraries of bases.

My own peculiar base functions tend to be very natural-looking, as they were originally honed for the modeling of natural phenomena such as mountains, clouds, and water. They thus tend to generate images that look like they were executed in a natural medium, such as oil paint. In fact, one of my main motivations in going into this area was to automate the generation of painterly textures such as that seen in Figure 15.17. Examples of my system's output are seen in Figures 19.4–19.9.

One nice thing is that the longer you work with a genetic program, the more "evolved" the results become. That is because as you accumulate a library of "fit" individuals, they can trade genetic information via sexual reproduction. Indeed, one feature of my program is being able to have individuals breed with the entire population of saved genomes, in a sort of orgiastic exchange of genetic information. An effect of this continued evolution is that subtrees become genes in their own right, being swapped in whole in the breeding process. Thus the "library" of genes can grow both in size and complexity as evolution proceeds. (We are now referring to a single function—what computer scientists would call a "primitive" or "atomic" function—as a base and a tree of any size as a gene.)

OTHER EXAMPLES OF GENETIC PROGRAMMING AND GENETIC ART

Karl Sims has taken this paradigm of genetic programming and genetic art farther than the rest of us. For example, he evolves the behaviors of three-dimensional textures (Sims 1991) and virtual creatures (Sims 1994) (see *www.biota.org/ksims.html*). William Latham (Todd and Latham 1993) has done some remarkable work in a system designed to generate 3D sculptures that can bear uncanny resemblances to creatures from the Cambrian epoch of life on Earth (*www. artworks.co.uk/*). Roman Verostko (*www.verostko.com*) employs the related concept of *epigenesis,* or the unfolding of form in the phenotype in the process of growth, in his plotter-generated artworks. Eric Wenger's ArtMatic product (*www.artmatic.com*) lets you play with a version of genetic textures.



FIGURES 19.4–19.9 Images generated by the genetic program Dr Mutatis. Note the rich visual complexity that arises through the automated evolution of a procedural texture. Copyright © F. Kenton Musgrave.

This is by no means an exhaustive listing of artists and scientists working in this exciting area. A search on the Web will turn up thousands of relevant sites. A good general guide is Linda Moss's site (*www.marlboro.edu/~lmoss/planhome*), which even offers source code. The bible of genetic programming is Koza (1992).

A FINAL DISTINCTION: GENETIC PROGRAMMING VERSUS GENETIC ALGORITHMS

In the literature, you'll read of genetic programming and genetic algorithms. The former is what I've described here, wherein the very code of the program that generates

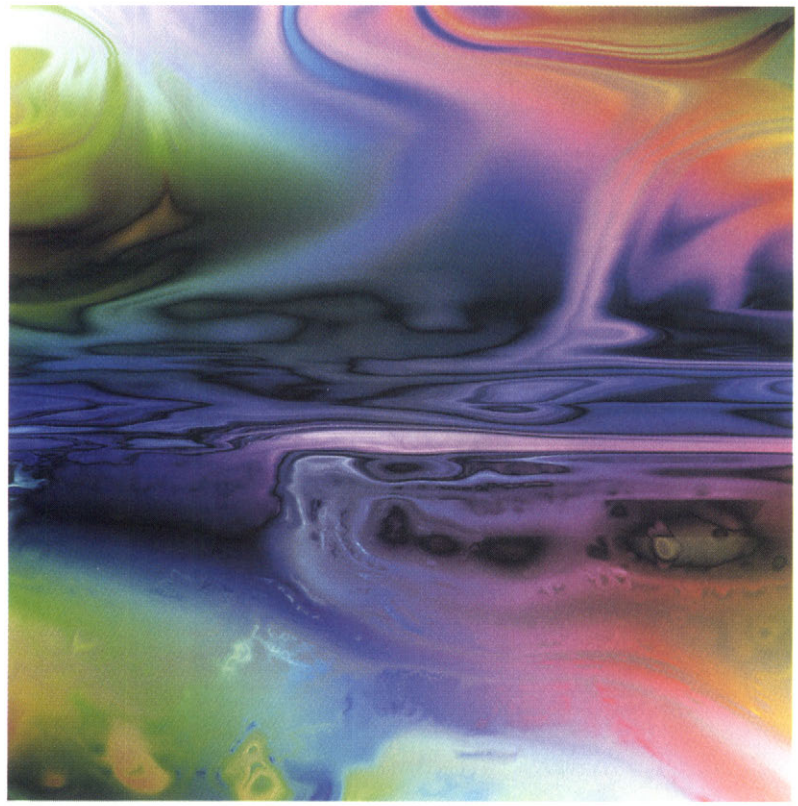


FIGURE 19.5

the phenotype is itself transforming over time. The latter is a little different: it assumes a fixed value on *n* for the *n*-space it explores and is thus perhaps more closely related to optimization strategies than to free-form artificial evolution. That is, it only searches the *n*-space for local aesthetic maxima, while genetic programs simultaneously define and search their *n*-space, with the value of *n* constantly changing. Thus they tend to be simultaneously more chaotic, hard to control, and productive—a familiar set of characteristics among creative people! At any rate, it may be helpful to be aware of the difference, so I'm pointing it out here.

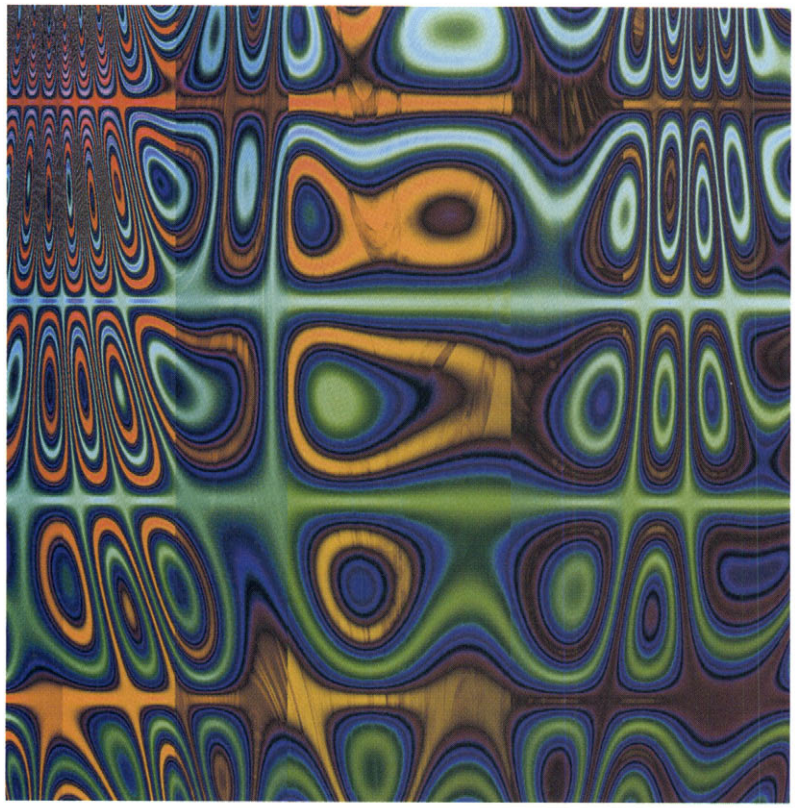


FIGURE 19.6

CONCLUSION

Genetic programming is one cool paradigm. It is amazingly automatic, has the world's best user interface—simply point and click on what you like—and it takes proceduralism to its logical end. While it may not be terribly *useful* because it's so hard to control and direct, it certainly is fun to play with, and it does create some striking images at times. It also gives the computer the greatest role in the creation of digital art of any paradigm I know. This is exciting in itself, as the computer can be a very capable, if simple-minded and cranky, artistic assistant.

Someday a program or programs will bring "genetic textures to the people." One fun thing about this prospect is that people could start trading genomes on the

FIGURE 19.7

Internet, thus accelerating evolution by cross-breeding between populations that spend most of their time in isolation on a given user's machine. This is analogous in nature to the evolutionary divergence of populations isolated on islands, which are occasionally intermingled by migration, chance travel, or formation of a land bridge.

Such universal exchange of genetic information will require standard encoding and interpretation machinery, just as DNA from some extraterrestrial organism would have zero chance of intermingling with that of life on Earth. Again, the genotype is like the operating system, and the cell is like the computer it runs on. Genotypes and operating systems are not highly portable. I foresee that, should we succeed in bringing genetic art to the people, we will all suffer for some time to come for a lack of foresight in the design of a robust and flexible system. I hope that one

FIGURE 19.8

day we'll be able to "breed" entire galaxies—indeed, a whole universe—of procedural planets, replete with automatic level of detail, in a future version of MojoWorld, which we'll get into in the next chapter. But that will be a few years from now, at least. We'll need a *lot* more processor power before we can deploy the powerful but expensive technique of genetic programming in that context.

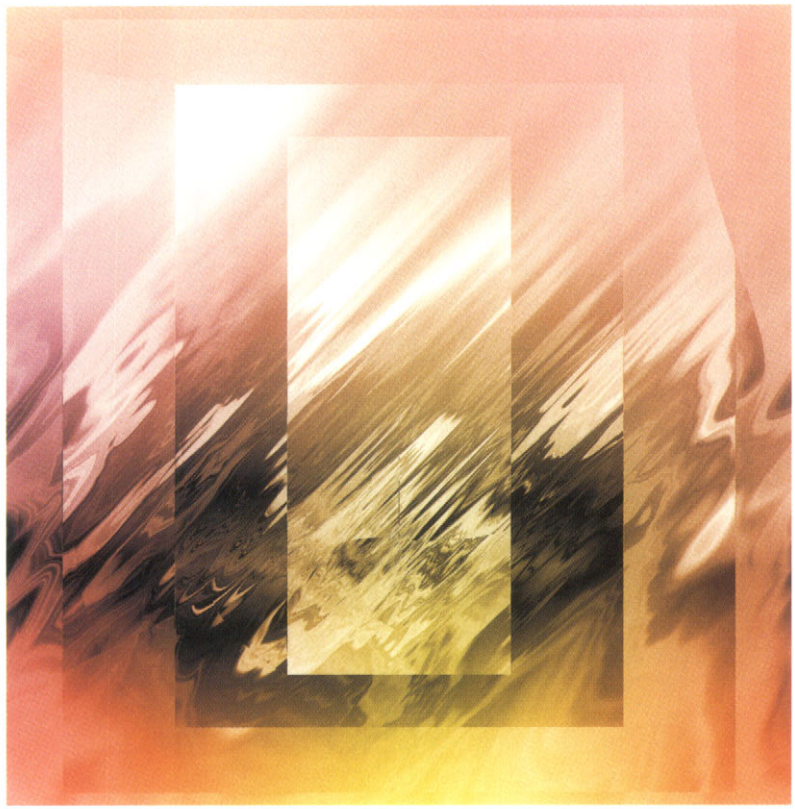


FIGURE 19.9