Visual Computer

Interactive evolution of equations for procedural models

Karl Sims

Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142, USA

This paper describes how the evolutionary mechanisms of variation and selection can be used to "evolve" complex equations used by procedural models for computer graphics and animation. An interactive process between the user and the computer allows the user to guide evolving equations by observing results and providing aesthetic information at each step of the process. The computer automatically generates random mutations of equations and combinations between equations to create new generations of results. This repeated interaction between user and computer allows the user to search hyperspaces of possible equations without being required to design the equations by hand or even understand them. Three examples of these techniques have been implemented and are described: procedurally generated pictures and textures, three-dimensional shapes represented by parametric equations, and two-dimensional dynamical systems described by sets of differential equations. It is proposed that these methods have potential as powerful tools for exploring procedural models and achieving flexible complexity with a minimum of user input and knowledge of details.

Key words: Evolution – Genetic algorithms – Procedural models

1 Introduction

Procedural models are increasingly employed in computer graphics to create scenes and animations having high degrees of complexity. A price paid for this complexity is that the user often loses the ability to maintain sufficient control over the results. Procedural models can also have limitations because the details of the procedure must be conceived, understood, and designed by humans. The techniques presented here contribute towards solutions to these problems by enabling "evolution" of procedural models using interactive "perceptual selection." Although they do not give complete control over every detail of the results, they do permit the creation of a large variety of complex entities that are still user directed, and the user is not required to understand the underlying equations involved.

Many years ago Charles Darwin proposed the theory that all species came about via the process of natural evolution (Darwin 1859). Evolution is now considered not only powerful enough to bring about biological entities as complex as humans and consciousness, but also useful in simulation to create algorithms and structures of higher levels of complexity than could easily be built by design. Genetic algorithms have proven useful for searching large spaces using simulated systems of variation and selection (Goldberg 1989; Grenfenstette 1985, 1987; Schaffer 1989). In *The Blind Watchmaker*, Dawkins has demonstrated the power of Darwinism with a simulated evolution of 2D branching structures made from sets of genetic parameters. The user selects the "biomorphs" that survive and reproduce to create each new generation (Dawkins 1986). Latham and Todd have applied these concepts to help generate computer sculptures made with constructive solid geometry techniques (Haggerty 1991; Todd and Latham 1991).

Variations on these techniques are used here with the emphasis on the potential of creating textures, objects, and motions that are useful in the production of computer graphics and animation, and also on the potential of using representations that are not bounded by a fixed space of possible results. The results presented here regarding evolution of textures and of dynamical systems can be found in additional publications in Sims 1991a, b.

1.1 Evolution

Both biological and simulated evolutions involve the basic concepts of genotype and phenotype, and

The Visual Computer (1993) 9:466–476

the processes of expression, selection, and reproduction with variation.

The *genotype* is the genetic information that codes for the creation of an individual. In biological systems, genotypes are normally composed of DNA. In simulated evolutions there are many possible representations of genotypes, such as strings of binary digits, sets of procedural parameters, or symbolic expressions. The *phenotype* is the individual itself, or the form that results from the developmental rules and the genotype. *Expression* is the process by which the phenotype is generated from the genotype. For example, expression can be a biological developmental process that reads and executes the information from DNA strands, or a set of procedural rules that use a set of genetic parameters to create a simulated structure. Usually, there is a significant amplification of information between the genotype and phenotype.

Selection is the process by which the fitness of phenotypes is determined. The likelihood of survival and the number of new offspring an individual generates is proportional to its fitness measure. *Fitness* is simply the ability of an organism to survive and reproduce. In simulation, it can be calculated by an explicitly defined fitness evaluation function, or it can be provided by a human observer as it is in this work.

Reproduction is the process by which new genotypes are generated from an existing genotype or genotypes. For evolution to progress there must be *variation* or mutations in new genotypes with some frequency. Mutations are usually probabilistic as opposed to deterministic. Note that selection is, in general, nonrandom and is performed on phenotypes, but variation is usually random and is performed on the corresponding genotypes (Fig. 1). The repeated cycle of reproduction with variation and selection of the most fit individuals drives the evolution of a population towards higher and higher levels of fitness.

Sexual combination can allow genetic material of more than one parent to be mixed together in some way to create new genotypes. This permits features to evolve independently and later be combined into an individual genotype. Although it is not necessary for evolution to occur, it is a valuable practice that can enhance progress in both biological and simulated evolutions.

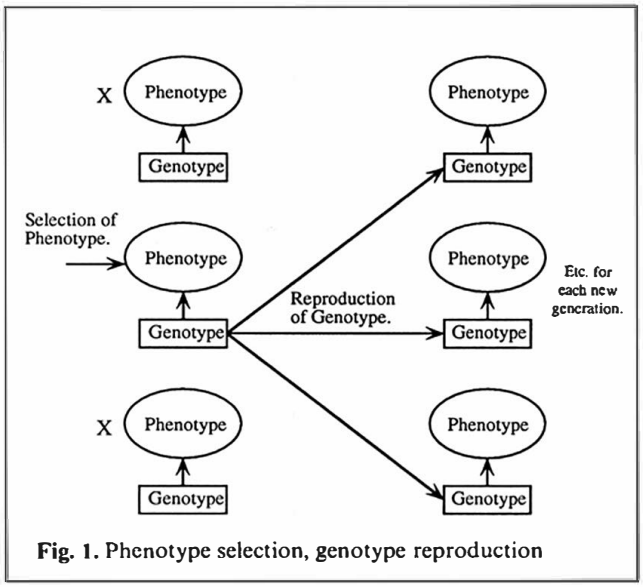


Fig. 1. Phenotype selection, genotype reproduction

1.2 Genetic algorithms

Genetic algorithms were first developed by Holland (1975) as robust searching techniques in which populations of test points are evolved by random variation and selection. They have become widely used in a number of applications to find optima in very large search spaces (Grenfenstette 1985, 1987; Schaffer 1989).

Genetic algorithms differ from the examples presented in this paper in that they usually use an explicit analytic function to measure the fitness of phenotypes. Since it is difficult to measure the aesthetic visual success of simulated objects or images automatically, here the fitness is provided interactively by a human user based on visual perception.

Population sizes used for genetic algorithms are usually fairly large (100 to 1000 or more) to allow searching of many test points and surpassing local optima. At each generation, many individuals survive and reproduce to create the next generation. For the examples presented in this paper, the success of a solution is dependent on human opinion; therefore there is no single global optimum. Many local optima are potentially interesting solutions. For this reason, and also because of user interface practicality, a smaller population size has been used (4–20), and only one or two individuals are chosen to reproduce for each new generation.

Genotypes used in genetic algorithms traditionally consist of fixed-length character strings used by fixed expression rules. This is appropriate for

searching predefined dimensional spaces for optimum solutions, but these restrictions are sometimes limiting. Koza (1992) has used hierarchical lisp expressions as genotypes such that the dimensionality of the search space itself can be extended to solve problems such as artificial ant navigation and game strategies successfully. Discovery systems, such as AM and Eurisko, also use a form of mutating lisp programs (Lenat and Brown 1984). The examples presented here also use genotypic representations composed of lisp expressions, although the set of functions used includes various vector transformations, noise generators, and image processing operations, as well as standard numerical functions.

In the next section, techniques for generating, mutating, and combining symbolic lisp expressions to explore hyperspaces of possible equations are described. In Sect. 3, 4, and 5, examples are presented that use these techniques to evolve 2D textures, 3D parametric objects, and 2D dynamical systems. Finally, results are discussed, suggestions are made for future work, and conclusions are given.

2 Lisp expressions as genotypes

Traditional genotypes that use fixed-length strings of parameters or digits and fixed expression rules are limited by having solid boundaries on the set of possible phenotypes. There is no possibility for the evolution of a new developmental rule or a new parameter. There is no way for the genetic space to be extended beyond its original definition – the N-dimensional genetic space will remain only N-dimensional. To surpass this limitation, it is desirable to include procedural information in the genotype instead of just parameter data, and the procedural and data elements of the genotype should not be restricted to a specific structure or size.

Hierarchical lisp expressions are used as genotypes in an attempt to meet these needs. A set of lisp functions and a set of argument generators are used to create arbitrary expressions that can be mutated, evolved, and evaluated to generate phenotypes. Some mutations can create larger expressions with new parameters and extend the space of possible phenotypes, while others just adjust existing parts of the expression.

Equations used by procedural models can often be represented by one or more lisp expressions. These expressions or sets of expressions become the genotypes for evolvable procedural models. For example, a texture-generating procedure can be described by an expression that calculates a color for each pixel coordinate (X, Y), and a procedure for generating a 3D parametric surface can be described by an equation that calculates a 3D vertex location for each parameteric variable pair (U, V).

For each application, a *function set* defines a set of primitive operations from which lisp expressions can be assembled. A basic function set might consist of some simple common lisp operations: +, −, *, /, *mod*, *round*, *min*, *max*, *abs*, *expt*, *log*, *sin*, and *cos* (Steele 1984). Function sets can be extended or adjusted to give various hyperspaces of possible results.

To begin a session of interactive evolution, an initial population of genotypes consisting of simple randomly generated lisp expression is created. These are assembled by first choosing a function at random from the function set and then generating as many arguments at random as that function requires. Arguments can be of several types: constant scalar values, three element vectors, variables such a X or Y pixel coordinates or additional random expressions generated recursively. The initial genotypes of the population are then expressed by performing the calculations described in their lisp expressions, and the resulting phenotypes are displayed to the user for interactive selection. The best individuals are then selected to survive and reproduce to create the next generation, and the process repeats with the new population.

2.1 Mutating symbolic expressions

Symbolic expressions must be reproduced with mutations for their evolution to progress. There are several properties of symbolic expression mutation that are desirable. Expressions should often be only slightly modified, but sometimes significantly adjusted in structure and size. Large random changes in genotype usually result in large jumps in phenotype that are less likely to be improvements, but are necessary for extending the expression to more complex forms.

A recursive mutation scheme is used to mutate expressions. Lisp expressions are traversed as tree

structures, and each node is in turn subject to possible mutations. Each type of mutation occurs at different frequencies, depending on the type of node:

1. Any node can mutate into a new random expression. This allows for large changes and usually results in a fairly significant alteration of the phenotype.
2. If the node is a scalar value, it can be adjusted by the addition of some random amount.
3. If the node is a vector, it can be adjusted by adding random amounts to each element.
4. If the node is a function, it can mutate into a different function. For example (*absX*) might become (*cosX*). If this mutation occurs, the arguments of the function are also adjusted if necessary to the correct number and types.
5. An expression can become the argument to a new random function. Other arguments are generated at random if necessary. For example, *X* might become (∗ *X .3*).
6. An argument to a function can jump out and become the new value for that node. For example, (∗ *X .3*) might become *X*. This is the inverse of the previous type of mutation.
7. Finally, a node can become a copy of another node from the parent expression. For example, (+ (*absX*)(∗ *Y.6*)) might become (+ ((*abs* (∗ *Y.6*))(∗ *Y.6*)). This causes effects similar to those caused by mating an expression with itself. It allows subexpressions to duplicate themselves within the overall expression.

Other types of mutations could certainly be implemented, but these are sufficient for a reasonable balance of slight modifications and potential for changes in complexity. Figure 2 shows a parent object in the upper left with 19 offspring created by random mutations of its equations. Note that some mutations cause little or no variation from the parent, while others cause significant alterations.

It is preferable to adjust the mutation frequencies such that a decrease in complexity is slightly more probable than an increase. This prevents the expressions from drifting towards large and slow forms without necessarily improving the results. They should still easily evolve towards larger sizes, but a larger size should be due to selection of improvements instead of random mutations with no effect. The overall mutation frequency is scaled inversely in proportion to the length of the parent expression. This decreases the probability of mutation at each node when the parent expression is large so that some stability of the phenotypes is maintained.

The evaluation of expressions and display of the resulting images can require significant calculation times as expressions increase in size. To keep image evolution at interactive speeds, estimates of compute speeds are calculated for each expression by summing precomputed runtime averages for each function. Slow expressions are eliminated before ever being displayed to the user. New offspring with random mutations are generated and tested until satisfactory expressions occur. If necessary, this technique could also be performed to keep memory usage to a minimum.

2.2 Mating symbolic expressions

Symbolic expressions can be reproduced with sexual combinations to allow characteristics from separately evolved individuals to be mixed into a single individual. A node in the expression tree of one parent is chosen at random and replaced by a node chosen at random from the other parent. The new expression is checked for legal syntax and if necessary more matings are performed until a legal expression results. This *crossing over* technique allows any part of the structure of one parent to be inserted into any part of the other parent and allows two subexpressions that have evolved independently to be combined into one genotype.

2.3 Genetic cross dissolves

Another technique for combining expressions is achieved by performing "genetic cross dissolves" between two expressions. A new expression is created by copying the nodes of the original expressions where they are identical but interpolating between the nodes where they are different. Results of differing expression branches are first calculated and interpolated and then used by the remaining parts of the expression. If the two expressions have different root nodes, the dissolve will cause a "fade" from one to the other, but if only parts within their structures are different, interesting transformations can occur as the interpolation proceeds. For example, the two simple expressions (∗ *X Y*) and (∗ *X .6*) could be dissolved to give the

expression (* X (*dissolve* Y .6 α)) where α is varied to perform the interpolation.
This technique uses the existing genetic representation of evolved procedural models to generate in-betweens for a smooth transition from one to another. It is an example of the usefulness of an alternate level of control given by the underlying genetic information. A series of frames from a genetic cross dissolve between two parametric objects is shown in Fig. 3.
Repeated interpolations can be a useful method for creating animation from a series of evolved structures. "Genetic splines" can even be used to give smoother interpolation between multiple control genotypes.
The following sections will present three specific examples of procedural models the equations of which are represented as lisp expressions so they can be mutated, mated, and interactively evolved.

3 Evolving pictures and textures

The first example involves the procedural generation of textures by symbolic expressions that describe color as a function of the X and Y pixel coordinates:

$$C_{(R,G,B)} = F(X, Y)$$

Equations that perform this mapping are evolved using a *function set* containing vector transformations, procedural noise generators, and image processing operations, as well as some standard common lisp functions:

+, −, *, /, *mod*, *round*, *min*, *max*, *abs*, *expt*, *log*, *and*, *or*, *xor*, *sin*, *cos*, *atan*, *if*, *dissolve*, *hsv-to-rgb*, *vector*, *transform-vector*, *bw-noise*, *color-noise*, *warped-bw-noise*, *warped-color-noise*, *blur*, *band-pass*, *grad-mag*, *grad-dir*, *bump*, *ifs*, *warped-ifs*, *kaleidoscope*, *warp-abs*, *warp-rel*, *warp-by-grad*.

Each function takes a certain number of arguments and calculates and returns an image of scalar (b/w) or vector (rgb color) values.
Noise generators can create solid scalar and vector noise at various frequencies with random seeds passed as arguments so that specific patterns can be preserved between generations (Fig. 4f; Lewis 1989). The warped versions of functions take input coordinates as arguments instead of pixel coordinates, allowing the result to be distorted by an arbitrary inverse mapping function (Fig. 4h). Boolean operations (*and*, *or*, and *xor*) operate on each bit of floating-point numbers and can cause fractal-like grid patterns (Fig. 4e). Versions of *sin* and *cos* which normalize their results between 0.0 and 1.0 instead of −1.0 and 1.0 can be useful. Some functions such as blurs, convolutions, and those that use gradients also use neighboring pixel values to calculate their results (Fig. 4g). *Band-pass* convolutions are performed using a difference of Gaussians filter which can enhance edges. Iterative function systems (*ifs*) can generate fractal patterns and shapes, and a *kaleidoscope* function generates triangular patterns.
It might be interesting to include many other functions in this *function set*, but those given have provided for a fairly wide variety of resulting images. Details of the specific implementations of each function are not given here because they are not as important as the evolution process itself. Most of the functions have been adapted either to coerce the arguments into the required types, or to perform differently according to the argument types given to them. Arguments to certain functions can be restricted optionally to some subset of the available types. For the most part these functions receive and return images, and could be considered as image processing operations. Expressions made from these functions are simply evaluated to produce images. Figure 4 shows examples of some simple expressions and their resulting images.
Interactive evolution of these expressions is performed by first generating a population of simple random expressions and displaying them to the user for selection. The expressions of images selected by the user are reproduced with mutations for each new generation such that more and more complex expressions and more perceptually interesting images can occur. Figure 5 was generated from this expression:

(*sin* (+ (− (*grad-direction* (*blur* (*if* (*hsv-to-rgb* (*warped-color-noise* (*vector* 0.57 0.73 0.92) (*/* 1.85 (*warped-color-noise X Y* 0.02 3.08)) 0.11 2.4)) (*vector* 0.54 0.73 0.59) (*vector* 1.06 0.82 0.06)) 3.1) 1.46 5.9) (*hsv-to-rgb* (*warped-color-noise Y* (*/* 4.5 (*warped-color-noise Y* (*/ X Y*) 2.4 2.4)) 0.02 2.4))) *X*))

Note that expressions only five or six lines long can generate images of fair complexity. Fortunately, analysis of expressions is not required when using these methods to create them. Users usually stop attempting to understand why each expres-

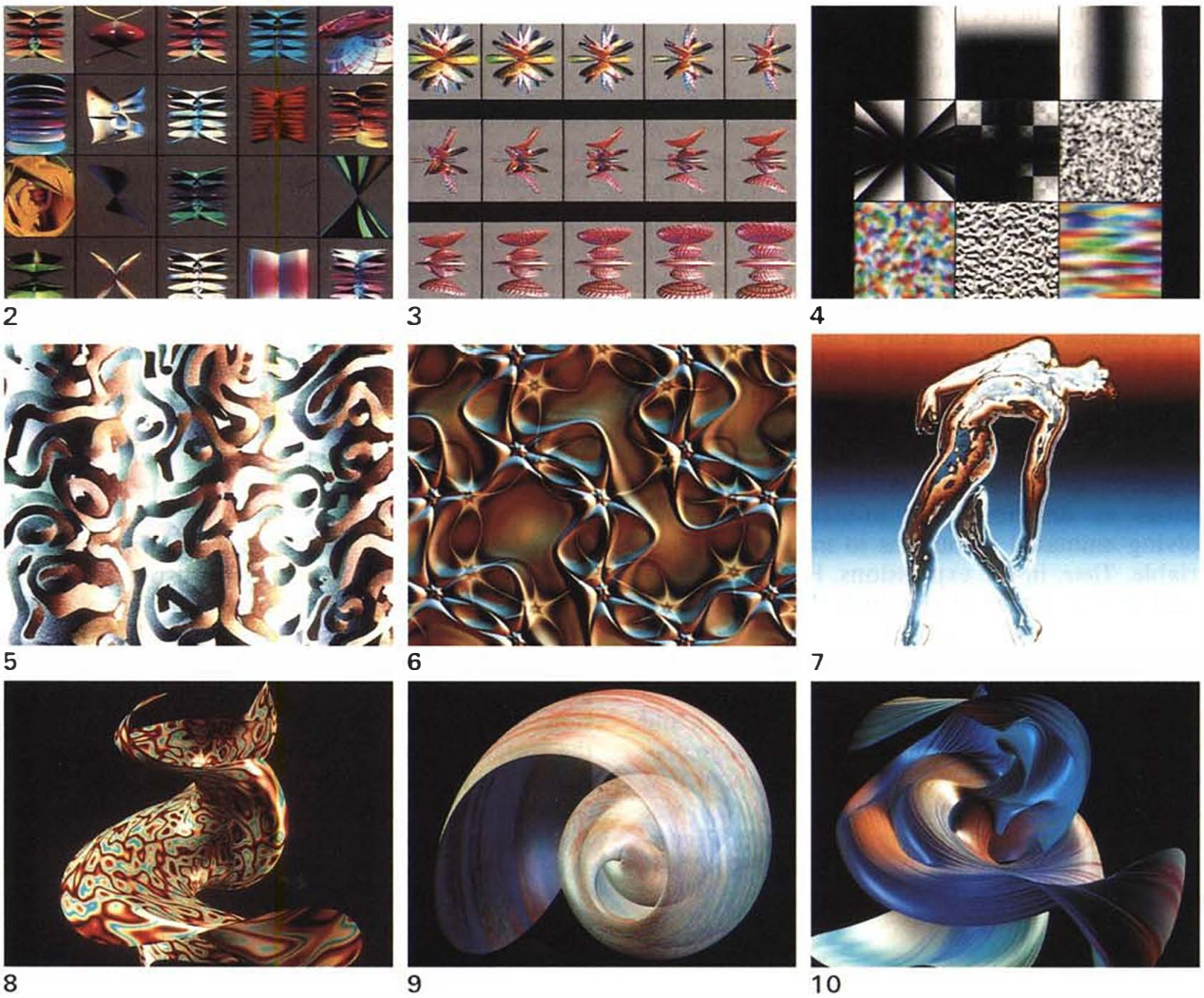


Fig. 2. Parent and 19 mutations

Fig. 3. Frames from a "genetic cross dissolve"

Fig. 4a–i. Simple image expression examples reaching *left* to *right*, *top* to *bottom*: a *X*; b *Y*; c (abs *X*); d (mod *X* (abs *Y*)); e (and *X Y*); f (bw-noise .2 2); g (color-noise .1 2); h (grad-direction (bw-noise .15 2) .0 .0); i (warped-color-noise (* *X* .2) *Y* .1 2)

Fig. 5. Evolved texture

Fig. 6. Frame from "Primordial Dance"

Fig. 7. Processed photograph

Fig. 8. Parametric surface

Fig. 9. Shell shape

Fig. 10. Folded structure

sion generates each image. Figure 6 was also generated in this way. Its corresponding expression, and others, are included in the appendix.

3.1 Volume textures, animation, and image processing

Other procedural models can be evolved with variations on the methods already described. Volume textures, animated textures, and image processing functions can be represented by adding to the set of input variables that are included in the expressions.

Volume textures can be described by adding a third variable, *Z*, to the list of available arguments. This enables functions to evolve that calculate colors

for each point in (X,Y,Z) space. The *function set* is adjusted for better results; 2D functions that require neighboiring pixel values such as convolutions and warps are removed, and 3D solid noise generating functions are added. These expressions are more difficult to visualize because they encompass all of 3D space. They are evaluated on the surfaces of spheres and planes for fast previewing and selection. Evolved volume expressions can then be incorporated into procedural shading functions to texture arbitrary objects. This process allows complex volume textures such as those described in Peachy (1985) and Perlin (1985) to be evolved without requiring specific equations to be understood or carefully adjusted by hand.

Animations of textures can be created by performing *genetic cross dissolves* between evolved static textures as described, but another method for evolving animated textures is to include an input variable, *Time*, in the expressions. Functions of X, Y, and *Time* can then evolve such that moving images are produced when the value of *Time* is smoothly varied.

An input image can also be added to the list of available arguments to make functions of X, Y, and *Image*. This allows creation of complex image processing and warping functions that compute new images from given input images. Figure 7 was created from an input image of a human figure. The short film *Primordial Dance* (Sims 1991c) was created using a variety of these methods.

4 Evolving 3D shapes

The second example of interactive evolution involves a procedural model for creating 3D parametric surfaces. The shape of a surface is described by an arbitrary function that calculates a 3D position as a function of two parametric variables U and V:

$$P_{(X,Y,Z)} = F(U, V)$$

The form of this equation is somewhat similar to that of the first application, except U and V are the input variables to the expressions instead of X and Y, and the expressions calculate 3D coordinates instead of colors. The *function set* normally used here avoids functions that cause discontinuities, but includes various functions for transforming between coordinate systems and performing 3D manipulations and distortions:

*+, −, *, /, min, max, abs, negate, sin, cos, sqrt, square, expt, dissolve, vector, noise, warped-noise, cyl-to-xyz, sph-to-xyz, rotate, vortex.*

As examples, the expression (*vector U V 0*) would create a simple flat grid on the X, Y plane, and the expression (*sph-to-xyz* (*vector 1.0 U V*)) would produce a unit sphere. A fairly wide variety of complex shapes can be described by longer expressions assembled from this set of functions. The shape of the object in Fig. 8 was interactively evolved, and was generated by the following expression:

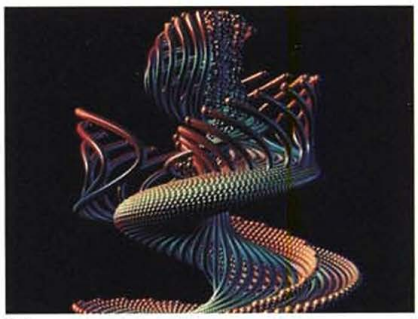
(*rotate* (*cyl-to-xyz* (*vector* (− −0.19 *U*) *V V*)) (* (*vector* −0.007 0.4 0.87) *U*) (*noise* −0.25 1.37))

The shapes in Figs. 9–11 were also created in this way and the expressions that generated them are listed in the appendix.

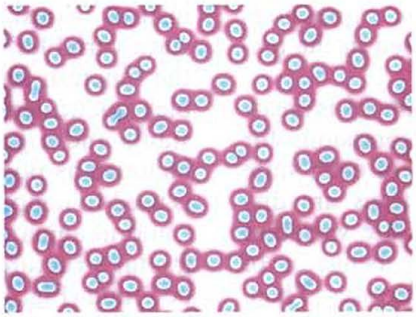
Procedural textures for the surfaces of parametric shapes can also be described by lisp expressions and included in these genotypes. An expression that calculates a surface color for each U, V is evaluated at each surface element during the rendering process. These are similar to the texture expressions already described, but they are evaluated on the surfaces of objects instead of on the flat image plane. Textures of surfaces can be evolved either simultaneously with the shape or independently. Both the shape expression and the texture expression are subjected to mutations during reproduction, but either can be made stable by selective mutation prevention. The procedurally generated textures and colors of the objects in Figs. 8–11 were interactively evolved.

Parametric surfaces are polygonalized for rendering by evaluating the expression at small regular intervals within some range of the parametric variables, such as 128×128 discrete samples of U and V between -1.0 and 1.0. The shapes shown were polygonalized and rendered in this way. A method that generates polygons or patches more adaptively, or even renders the parametric surface directly, might be helpful, but this simple approach was chosen for its ease of implementation.

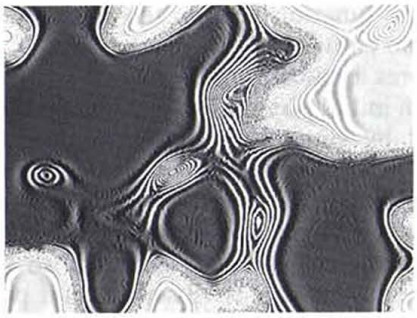
An alternate method of rendering parametric surfaces is to create a small sphere at each U, V sample as shown in Fig. 11. This allows the structure of discontinuous or tangled shapes to be visualized and is sometimes advantageous. When using this rendering method, discontinuous functions such as *mod*, *round*, *and*, *or*, and *xor* can be added back to the function set.



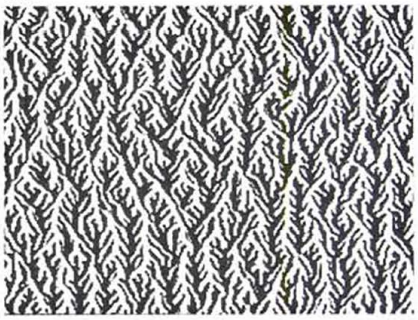
11



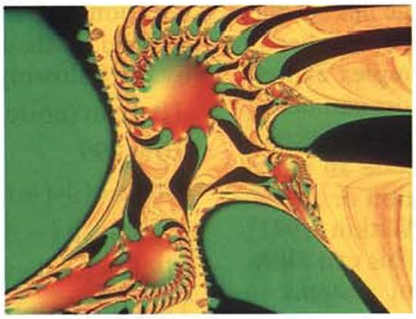
12



13



14



15

Fig. 11. Visualizing with spheres

Fig. 12. Cell shapes

Fig. 13. Wave generators

Fig. 14. Branching patterns

Fig. 15. Fractal structures

In some cases the polygonalization and rendering of complex parametric surfaces can not be performed in real-time to allow as much interactive speed as might be desired. A quick display of vertex points can often give enough initial information about the shapes to make selections and allow efficient interactivity.

5 Evolving dynamical systems

The third and final example of interactive evolution of procedural models involves 2D dynamical systems described by systems of equations. In this application, several cooperating lisp expressions are used to determine the initial states and time derivatives of state variables of dynamical systems. For example, a system containing two quantities, A and B, at each grid location is described by four equations:

$A_0 = F_{A0}(X, Y)$

$B_0 = F_{B0}(X, Y)$

$dA/dt = F_{dA}(A, B)$

$dB/dt = F_{dB}(A, B)$

F_{A0} and F_{B0} are functions that determine the initial values for each element of A and B from their grid coordinates (X,Y). F_{dA} and F_{dB} are functions that determine the rate of change for each element of A and B using the current state of the system. Arbitrary functions for F_{A0}, F_{B0}, F_{dA}, and F_{dB}, are specified by lisp expressions that can vary in size, structure, and behavior. A genotype contains one lisp expression for each of these functions. For example, a genotype that would describe a simple reaction-diffusion-like system of two chemicals that diffuse and inhibit each other might be:

$A_0 =$ (noise .82)

$B_0 =$ (noise .93)

$dA/dt =$ (− (laplacian A) B)

$dB/dt =$ (− (laplacian B) A)

The set of functions used to compose these lisp expressions contains the usual common lisp functions, but also contains operations that can perform various convolutions, and find first- and second-order spatial derivatives:

+, −, *, /, *mod*, *round*, *min*, *max*, *abs*, *expt*, *log*, *sin*, *cos*, *atan*, *negate*, *sqrt*, *square*, *dissolve*, *if-plusp*, *x-grad*, *y-grad*, *grad-mag*, *grad-in-direction*, *grad-direction*, *neighbor-min*, *neighbor-max*, *neighbor-ave*, *convolve-with-mask*, *curl*, *laplacian*, *anisotropic-laplacian*.

The function set for the initial state expressions also contains a *noise* procedure, as used in the textures application above.

An initial population of dynamical systems is created by generating simple random expressions for the initial state and time derivatives of each state variable. The corresponding simulations are displayed to the user by mapping the state variables into colors for each iteration so the behavior of the system can be observed as it progresses. Then, the user selects one or more of these systems for mutation and/or mating to produce the next generation, and the process repeats. After a number of generations, genotypes with fairly complex expressions and interesting resulting behaviors can occur.

As an alternative to starting with randomly generated expressions, the user can hand-code an initial set of equations, such as a wave equation or a reaction-diffusion system (Turk 1991; Witkin 1991), and begin the evolution from there. This can allow unexpected variations of initial input systems to be explored.

For simplicity, simulations of continuous dynamical systems are performed using Euler's method of integration. The differential equations are approximated for a small discrete time interval Δt. For example,

$$\frac{dA}{dt} = F(A)$$

would be simulated by computing many discrete updates of the value of A:

$$A' = A + \Delta t F(A)$$

When Δt is smaller, the simulation is more accurate, but more computation is required. ($\Delta t = 0.1$ is often used.)

Systems can sometimes generate values that exceed the legal bounds of numerical representation. Values are regularly clamped to some legal bounds to avoid overflow errors. These particular discretizations of time and clamping parameters can affect the behaviors of some systems. In fact, systems that exploit these specific procedures for interesting effects sometimes evolve.

The first derivatives are also included as possible arguments in the expressions. Resulting behaviors might not be consistent if Δt is modified, but for a given time increment, this can help interesting physical-like systems to occur.

The space of possible dynamical systems can be further enhanced by allowing complex numbers, instead of just real values, to be included in the state variables and expressions. The operations in the function set are adjusted to perform on complex quantities as well as reals, and complex constants and a grid coordinate value, $\#C(X\ Y)$, are included as possible arguments. (The form $\#C(ri)$ is used to denote a complex quantity with real part r and imaginary part i.) Various spiral shapes and fractal structures that use complex arithmetic can be found (Fig. 15).

Figures 12–15 show the results after a number of iterations of some dynamical systems that were evolved by these methods. Figure 12 was produced by the following system of equations:

$A0 = (\sin\ (\text{noise}\ -.14\ -.77))$

$B0 = 1.99$

$dA/dt = (+\ (+\ (\text{laplacian A } 2.1)$

$\quad (\text{if-plusp}\ (-\text{A B})\ .4\ .0))(*\ -.38\ \text{A}))$

$dB/dt = (+\ (\text{laplacian A } 4.99)(*\ -.4\ \text{B}))$

This system proceeds from random noise towards a stable pattern of circular cell-like shapes. Again, it is often not obvious why a set of equations produces the behavior is does, even for relatively short expressions. Fortunately, a complete understanding of these equations is not required even by the creator. The expressions that specify the equations that produced Figs. 13–15 are given in the Appendix.

6 Results

The three examples of interactive evolution described have been implemented on the Connection Machine® system CM-2, a data parallel supercomputer (Hillis 1987). The parallel implementation details will not be discussed in detail here, but each application is reasonably suited for highly parallel representation and computation. Lisp expression mutations and combinations are performed on a *front-end* computer and the expressions for each application are evaluated on the Connection Machine system in parallel with one virtual processor per data element (pixels, coordinates, or grid cells). This usually allows calculation and display of results to be performed fast enough for efficient interactivity.

Many of the procedurally generated results shown here were evolved in timescales of only a small number of minutes – probably much faster than

they could be designed. These methods allow procedural models to be explored efficiently and with the very simple interface of just choosing from samples.

Two different approaches of user selection behavior are possible. The user can have a goal in mind and select samples that are closer to that goal until it is hopefully reached. Alternatively, the user can follow the more interesting samples as they occur without attempting to reach any specific goal. The latter approach often leads to more interesting results.

These various evolved products can be saved in the concise form of the final genotypic expression itself. This facilitates keeping large libraries of evolved forms which can then be used to contribute to further evolutions by mating them with other forms or further evolving them in new directions.

7 Future work

Many other procedural models could potentially be explored using these techniques. Other types of grammatical systems that describe generative processes for creating various entities could also be subjected to mutation, mating, and interactive selection. Procedures could be explored that control distributions or generate motions for many elements such as systems of particles or brush strokes. Algorithms that use rules to construct, arrange, and combine 3D geometric primitives could also be evolved. These tools might also be valuable in domains beyond computer graphics for the design of various items such as fonts, clothing, or even sounds.

Several variations on these methods for artificial evolution might make interesting experiments. One could attempt to automatically evolve symbolic expressions that could generate simple specific goals. A differencing function could be used to calculate a *fitness* based on how close a test genotype was to the goal, and the goal could be searched for by automatic selection. Then, interactive selection could be used to evolve further results starting from there.

Large amounts of information of all the human selection choices of many evolutions could be saved and analyzed. A difficult challenge would be to create a system that could generalize and "understand" what makes a result visually successful, and even generate other images that meet these learned criteria.

Combinations of random variations and nonrandom variations using learned information might be helpful. If a user picks phenotypes in a certain direction from the parent, mutations for the next generation might have a tendency to continue in that same direction, giving "evolutionary momentum."

Also, combinations of evolution and the ability to apply specific adjustments to the genotype might allow more user control of evolved results. Automatic "genetic engineering" could permit a user to make requests that might, for example, make an object more blue, or a texture more grainy.

8 Conclusion

Interactive evolution of equations for procedural models has been demonstrated to be a potentially powerful tool for the creation of textures, objects, and dynamical systems for use in computer graphics and animation. Reproduction with random variations and survival of the visually interesting can lead to useful results. Representations for genotypes that are not limited to fixed spaces and can grow in complexity have shown to be worthwhile.

Evolution is a method for creating and exploring complexity that does not require human understanding of the specific process involved. This process of interactive evolution could be considered a system for helping the user with creative explorations, or it might be considered a system which attempts to "learn" about human aesthetics from the user. In either case, it allows the user and computer to work together interactively in a new way to produce results that neither could easily produce alone.

An important limiting factor in the usefulness of interactive evolution is that samples need to be generated quickly enough that it is advantageous for the user to choose from random samples rather than to adjust new samples carefully by hand. The computer needs to generate and display samples fast enough to keep the user interested while selecting amongst them. As computation becomes more powerful and available, these methods should become advantageous in more and more domains.

Acknowledgments. Thanks to Lew Tucker, Gary Oberbrunner, Matt Fitzgibbon, and Jim Salem for help and CM graphics software support. Thanks to JP Massar for Starlisp support, and to Katy Smith for proofreading. Thanks to Pattie Maes for encouragement, to Richard Dawkins for demonstrating the

interactive concept, and to Peter Schröder for being a helpful early user of these tools.
A version of this paper was previously published in the Imagina Proceedings, Monte-Carlo, January 1992.

References

1. Darwin C (1859) The origin of species. New American Library, Mentor paperback, New York
2. Dawkins R (1986) The blind watchmaker. Harlow, London
3. Goldberg DE (1989) Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Rending, MA
4. Grenfenstette JJ (1985) Proc 1st Int Conf Genetic Algorithms Their Appl, Hillsdale, N.J.
5. Grenfenstette JJ (1987) Genetic algorithms and their applications. Proc Int Conf Genetic Algorithms, Hillsdale, N.J.
6. Haggerty M (1991) Evolution by esthetics, an interview with W. Latham and S. Todd. IEEE Comput Graph 11, pp 5–9
7. Hillis, WD (1987) The connection machine. Scientific American 256:108–115
8. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor, Mich.
9. Koza JR (1990) Genetic programming. The MIT Press, Cambridge, MA
10. Lenat DB, Brown JS (1984) Why AM and EURISKO appear to work. Artificial Intelligence 23:269–294
11. Lewis JP (1989) Algorithms for solid noise synthesis. Comput Graph 23:263–270
12. Peachy D (1985) Solid texturing of complex surfaces. Comput Graph 19:279–286
13. Perlin K (1985) An image synthesizer. Comput Graph 19:287–296
14. Schaffer JD (1989) Proc 3rd Int Conf Genetic Algorithms, Morgan Kaufman Publishers, San Mateo, CA
15. Sims K (1991a) Artificial evolution for computer graphics. Comput Graph 25:319–328
16. Sims K (1991b) Interactive evolution of dynamical systems Proc Eur Conf Artificial Life, Paris, MIT Press, pp 171–178
17. Sims K (1991c) Primordial Dance, Siggraph/ACM Video Review-Electronic Theatre
18. Steele G (1984) Common Lisp, The Language, Digital Press
19. Todd SJP, Latham W (1991) Mutator, a subjective human interface for evolution of computer sculptures. IBM United Kingdom Scientific Centre Report 248
20. Turk G (1991) Generating textures for arbitrary surfaces using reaction-diffusion. Comput Graph 25:289–298
21. Witkin A, Kass M (1991) Reaction diffusion textures. Comput Graph 25:299–308

Appendix

Figure 6, Frame from "Primordial Dance:"
(dissolve (/ (warped-bw-noise (expt (blur (kaleidoscope 2.1 6.9 *Y* (vector 0.96 0.42 0.51)) 7.6) 0.12) 0.5 0.17 9.6) (vector 0.49 0.53 0.92)) (color-grad (warped-bw-noise (expt (blur (kaleidoscope 1.6 6.9 *Y* (vector 0.30 0.45 −0.14)) 3.8) 0.12) (bump (ifs 4.5 0.82 0.22 3.8 0.45 0.049 3.7 16.1 −0.87 2.3 0.10 0.46 −5.0 7.6 12.6 0.05) 6.4 0.007 (vector 0.98 0.12 0.18) (vector 0.11 0.59 0.014) 1.07 8.1 12.1) 0.179.6) 2.8 2.0 (vector 0.47 0.04 0.22) 2.0) 0.48)

Figure 7, Processed photograph:
(cos (+ (+ (cos (+ (warp-by-grad Image Image 0.34 1.12) (vector .28 .40 .46))) *Y*) Image))

Figure 9, Shell shape:
(rotate (dissolve (cyl-to-xyz (vector −0.95 *U V*)) (vector 0.46 0.16 −0.33) (negate *U*)) (* (vector −0.006 −0.008 1.05) 2.29) *U*)

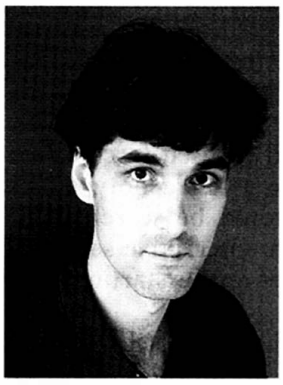
Figure 10, Folded structure:
(rotate (dissolve (cyl-to-xyz (vector (− −0.19 *U*) *V V*)) (rotate (vector −0.77 −0.11 0.017) (vector 1.8 0.42 2.2) (* (− *U* −1.9) 1.18)) (negate (warped-noise-0.07 *V* 0.43 −0.24))) (* (vector −0.007 0.4 0.87) 1.14) (noise −0.25 1.37))

Figure 11, Visualizing with spheres:
(negate (cyl-to-xyz (− (vector 0.027 0.28 *U*) (+ (rotate (vector −0.063 1.82 −0.21) (if-plusp *V* (vector 0.24 −1.15 −0.98) (warped-noise *V V* (warped-noise *U* (atan 0.45 *U*) −1.02 −0.011) 1.87 −0.097)) (mod *V* −1.38)) (vector (and (xor (− (/ *U U*) *U*) −0.95) (warped-noise −0.019 −1.0 1.84 −1.29)) *U* (square *U*))))))

Figure 13, Wave generators:
$A0 = 0.33$
$B0 = 0.27$
$C0 =$ (log (− 0.5 (grad-mag-squared (noise −0.2 −0.04))) (/ (noise 0.02 0.03) (noise −0.007 −1.4)))
$dA/dt = C$
$dB/dt =$ (anisotropic-laplacian (sin *A*) *A* 0.9 0.08)
$dC/dt =$ (neighbor-ave (atan dA/dt (Laplacian *B* 1.8)))

Figure 14, Branching patterns:
$A0 = Y$
$B0 = 1.0$
$C0 =$ (+ (negate (noise 0.12 1.9)) *Y*)
$dA/dt =$ (neighbor-max (neighbor-max *C*))
$dB/dt =$ (*x*-grad *C*)
$dC/dt =$ (neighbor-ave (grad-direction *B* 0.25))

Figure 15, Fractal structures:
$A0 = \#C(X\ Y)$
$dA/dt =$ (+ (/ (+ (square *A*) 1.0) *A*) (+ −0.7 (expt (max (max *A* (laplacian (log *A* #*C* (−1.2 −0.05)) 0.11)) #*C* (0.21 −0.12)) 3.5)))



KARL SIMS received a B.S. in Life Sciences from the Massachusetts Institute of Technology in 1984. After working at Thinking Machines Corporation for a year he returned to the Massachusetts Institute of Technology to study graphics and animation at the Media Laboratory and received an M.S. in Visual Studies in 1987. He then joined the production research team at Whitney/Demos Productions in California, and later became co-founder and director of research for Hollywood based Optomystic. He currently works once again at Thinking Machines Corporation as a research scientist and artist in residence. His recent works of animation include: "Particle Dreams", "Excerpts from Leonardo's Deluge", "Panspermia", "Primordial Dance", and "Liquid Selves".