Evolving L-Systems To Generate Virtual Creatures

Gregory S. Hornby and Jordan B. Pollack

DEMO Lab
Computer Science Department
Brandeis University
Waltham, MA 02454-9110
hornby,pollack@cs.brandeis.edu

Abstract

Virtual creatures play an increasingly important role in computer graphics as special effects and background characters. The artificial evolution of such creatures potentially offers some relief from the difficult and time consuming task of specifying morphologies and behaviors. But, while artificial life techniques have been used to create a variety of virtual creatures, previous work has not scaled beyond creatures with 50 components and the most recent work has generated creatures that are unnatural looking. Here we describe a system that uses Lindenmayer systems (L-systems) as the encoding of an evolutionary algorithm (EA) for creating virtual creatures. Creatures evolved by this system have hundreds of parts, and the use of an L-system as the encoding results in creatures with a more natural look.

Key words: Animation, Artificial Life, Representation, Intelligent agents, Lindenmayer Systems (L-systems)

1 Introduction

As computers become more powerful it is increasingly our design ability, rather than computing power, that limits the richness of virtual worlds. Evolutionary algorithms (EAs), a technique inspired by biological evolution, have shown much promise in automating the process of producing creatures for virtual environments, yet the most recent work in this area, [16,20] has produced ungainly creatures with less than 50 components. The asymmetries of these creatures is a result of using a direct encoding, an explicit encoding with a one-to-one mapping from genotypic encoding to creature-part. As direct encodings have no re-use, symmetries and regularities do not occur, except by chance, and evolved structures tend to be unnatural looking.

In this we return to the spirit of [23], in which a graph structure was used as a generative encoding for the creatures. A generative encoding is a developmental method for producing a structure using a set of grammatical re-writing rules or a procedural process, not unlike a computer program with re-usable sub-procedures. Designs produced by a generative encoding have fractal-like self-similarities, giving them an organic look, and have been shown to have better scaling properties than direct encodings [3,12]. Here we use Lindenmayer Systems (L-systems) [18] as a more powerful, and general purpose, generative encoding than that of [23] to achieve moving creatures with hundreds of parts whose structure is more natural looking than [16,20].

More common than co-evolving morphology and controller has been work evolving controllers for pre-specified morphologies. Control systems for these works has included stimulus-response rules [21,25], neural controllers [9], and genetic programs [8]. The controllers of the creatures in this work are oscillator circuits; each actuated joint is controlled by an oscillator with its own frequency and relative phase offset. To achieve controllers that are reactive to the environment joints can be controlled by recurrent neural networks, as with [23,16,20], by including neural-network construction commands into the encoding language [13].

In the following sections we first outline the design space and describe the components of our generative design system, then we present our results and finally close with a discussion and conclusion of our work.

2 Methods

The system for producing moving creatures consists of an algorithm for optimizing creature designs, an encoding to represent the creatures for the optimization algorithm, and a method of constructing creatures from their encoding.

Evolutionary algorithms are used as the optimization algorithm for producing the virtual creatures. EAs are a class of stochastic search and optimization techniques inspired by natural evolution, these include genetic algorithms [10], evolutionary strategies [2], genetic programming [17] and evolutionary programming [6]. An EA maintains a population of candidate solutions from which it performs search by iteratively replacing poor members of the population with individuals generated by applying variation to good members of the population.

L-systems, a set of grammatical rewriting rules developed to model the biological development of multi-cellular organism [18], are used as the encoding for the EA. Analogous to the concurrent division of cells in multi-cellular organisms, the rewriting rules of an L-system are applied in parallel to all characters in a string. By iteratively applying the set of rewrite rules a complex string is created from a succession of simpler ones. In this work we use a set of construction commands as the characters for the L-system so that the strings produced by the evolved L-system are a sequence of commands for constructing a creature.

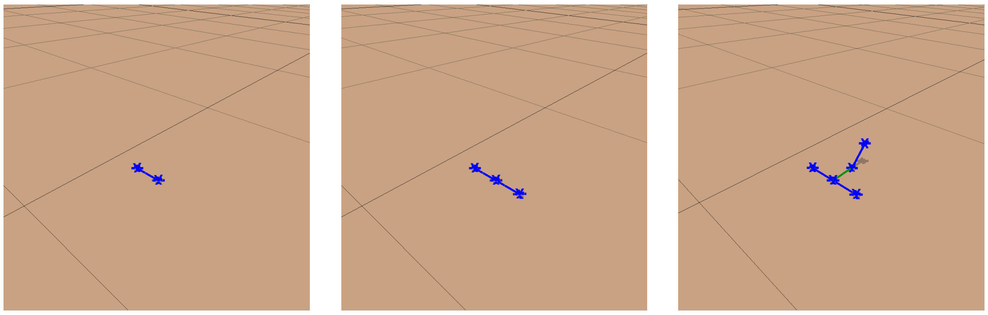
Once a string of construction commands is generated by an evolved L-system, it is passed to the creature constructor for evaluation. The creature constructor module follows the string of construction commands, building the creature piece by piece. After building the creature, it is evaluated for how well it moves and this score is passed back to the EA.

We now describe the method for constructing creatures, the L-system encoding and the EA in more detail.

2.1 Creature Constructor

Creatures are composed of bars that are connected by either fixed or actuated joints. The construction module builds creatures by processing a sequence of construction commands that specify how, and where, to attach bars to the existing design. This sequence of commands is based on the instruction language for a LOGO-style turtle [1]. As the turtle moves it creates bars, and commands in the language specify whether subsequent bars are to be attached with a fixed or actuated joint.

The turtle command language is similar to the L-system languages for creating plants [22]. *Push* and *pop* operators, '[' and ']', are used to store and retrieve the current state – consisting of the current location, orientation, and relative phase offset – to and from the stack. *Turn left/right/up/down/clockwise/counter-clockwise(n)* rotate the turtle's heading about the appropriate axis in units of 90°. To create bars for creatures, *forward(n)* moves the turtle forward in the current direction, creating a bar if none exists or traversing to the end of the existing bar and *backward(n)* goes back up the parent of the current bar. The joint commands move the turtle forward and end with a joint of the specified type which oscillates at a rate specified by the command's argument. By specifying the rate of oscillation and relative phase offset, a wide range of movement patterns can be generated. *Revolute-1(n)* creates a joint which oscillates from 0° to 90° about the Z-axis with speed n, *revolute-2(n)* creates a joint which oscillates from $-45°$ to 45° about the Z-axis with speed n, *twist-90(n)* creates a joint which oscillates from $0°$ to $90°$ about the X-axis with speed n, and *twist-180(n)* creates a joint which oscillates from $-90°$ to $90°$ about the X-axis with speed n. Combined with the rotation commands these allow actuated joints to be created that rotate about the primary axes. We also add a new command type, block repetition. Command sequences enclosed by '{' and '}' are repeated a number of times specified by the brackets' argument, thus { *forward(1)* }*(3)* is interpreted as: *forward(1) forward(1) forward(1)*.



(a) (b) (c)

Fig. 1. Building and simulating a 3D creature.

In figure 1 we show intermediate steps in the building of a creature as well as part of its animation. The single bar in figure 1.a is built from the string, *[left(1) forward(1)]*, and the two bar structure in figure 1.b is built from, *[left(1) forward(1)] [right(1) forward(1)]*. The final creature is made from the command sequence, *[left(1) forward(1)] [right(1) forward(1)] revolute-1(1) forward(1)*, and is shown in figure 1.c, where it is displayed part-way through its movement cycle.

Once an L-system specification is executed, and the resulting creature is constructed, its behavior is evaluated. In this work, creatures are evaluated in a quasi-dynamic simulator similar to that of [20]. As demonstrated in our previous 2D work [11], creatures designed in this simulator have the potential to transfer to the real world.

2.2 Parametric 0L-Systems

The main reason for using L-systems as the encoding for the evolutionary system is their ability to compactly describe more natural looking structures than the asymmetric structures produced with direct encodings [4,7]. As previous work evolving basic L-systems has tended to produce overly regular designs we used parametric L-systems (P0L-systems) [19] as a way to increase variation.

Formally, a P0L-system is defined as an ordered quadruplet, G = (V, Σ, ω, P) where, V is the alphabet of the system,

Σ is the set of formal parameters,

$\omega \in (V \times \Re^*)^+$ is a nonempty parametric word called the axiom, and

$P \subset (V \times \Sigma^*) \times C(\Sigma) \times (V \times \xi(\Sigma))^*$ is a finite set of productions.

The symbols : and $\rightarrow$ are used to separate the three components of a production: the predecessor, the condition and the successor. For example, a production with predecessor A(n0, n1), condition n1>5 and successor B(n1+1) c D(n1+0.5, n0-2) is written as:

$$A(n0, n1) : n1 > 5 \rightarrow B(n1+1)cD(n1+0.5, n0-2)$$

A production matches a module in a parametric word iff the letter in the module and the letter in the production predecessor are the same, the number of actual parameters in the module is equal to the number of formal parameters in the production predecessor, and the condition evaluates to true if the actual parameter values are substituted for the formal parameters in the production.

For implementation reasons we add constraints to our P0L-system. The condition is restricted to be comparisons as to whether a production parameter is greater than a constant value. Parameters to design commands are either a constant value or a production parameter. Parameters to productions are equations of the form: [*production parameter* | *constant*] [+ | $-$ | $\times$ | \ ] [*production parameter* | *constant*].

2.3 Evolutionary Algorithm

An evolutionary algorithm is used to evolve individual L-systems and the initial calling parameters of the first production rule along with the maximum number of iteration updates to be performed. The initial population of L-systems is created by making random production rules. Evolution then proceeds by iteratively selecting a collection of individuals with high fitness for parents and using them to create a new population of individual L-systems through mutation and recombination. As initialization, mutation and recombination are dependent on the encoding we describe them in greater detail.

An initial L-system is created from a blank template of a fixed number of production rules, each with a fixed number of condition-successor pairs. Conditions are created by randomly picking a parameter and a constant value to compare it against. Successor strings are created by stringing together sequences of randomly generated blocks of one to three characters, which may be enclosed by push and pop symbols, '[' and ']', or block replication symbols, '{' and '}'. Examples of initial blocks of characters are: *a(2.0) b(3.0) c(4.0)*, { *P2(n1+2.0,n1/3.0) d(3.0)* }*(2)*, and *[a(n0)]*. After an L-system is created, it is evaluated. L-systems that score below a preset threshold are discarded and a new one is randomly created in its place. This way the initial population consists of a variety of different solutions, each of which is a creature whose fitness is above the preset threshold.

Mutation creates a new individual by copying the parent individual and making a small change to it. First a production rule is selected at random from one of the used production rules and then this rule is changed in some way. Changes that can occur are: replacing one command with a random command; perturbing the parameter of a command by adding/subtracting a small value to it; changing the parameter equation to a production; adding/deleting a block of commands in a successor; or changing the condition equation.

For example, if the production $P0$ is selected to be mutated,

$$
\begin{aligned}
P0(n0, n1) : n0 > 5.0 &\rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1)\ c(3.0) \\
n0 > 2.0 &\rightarrow d(4.0)\ [\ P1(n1 - 1.0, n0/2.0)\ ]
\end{aligned}
$$

some of the possible mutations are,
Mutate an argument:

$$
\begin{aligned}
P0(n0, n1) : n0 > 5.0 &\rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1)\ c(3.0) \\
n0 > 2.0 &\rightarrow d(4.0)\ [\ P1(\mathbf{n1} - \mathbf{2.0}, n0/2.0)\ ]
\end{aligned}
$$

Delete random character(s):

$$
\begin{aligned}
P0(n0, n1) : n0 > 5.0 &\rightarrow \{\ a(1.0)\ \}(n1)\ c(3.0) \\
n0 > 2.0 &\rightarrow d(4.0)\ [\ P1(n1 - 1.0, n0/2.0)\ ]
\end{aligned}
$$

Insert a random block of 1-3 character(s):

$$
\begin{aligned}
P0(n0, n1) : n0 > 5.0 &\rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1)\ \mathbf{e}(\mathbf{4.0})\ c(3.0) \\
n0 > 2.0 &\rightarrow d(4.0)\ [\ P1(n1 - 1.0, n0/2.0)\ ]
\end{aligned}
$$

Recombination takes two individuals, $p1$ and $p2$, as parents and creates one child individual, c, by making it a copy of $p1$ and then inserting a small part of $p2$ into it. This is done by replacing one successor of c with a successor of

$p2$, inserting a sub-sequence of commands from a successor in $p2$ into c, or replacing a sub-sequence of commands in a successor of c with a sub-sequence of commands from a successor in $p2$.

For example if parent 1 has the following rule,

$$\begin{aligned} P3(n0, n1) : & \; n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1)\ c(3.0) \\ & \; n0 > 2.0 \rightarrow d(4.0)\ [\ P1(n1 - 1.0, n0/2.0)\ ] \end{aligned}$$

and parent 2 has the following rule,

$$\begin{aligned} P3(n0, n1) : & \; n1 > 3.0 \rightarrow b(3.0)\ a(2.0)\ c(1.0) \\ & \; n0 > 1.0 \rightarrow P1(n1 - 1.0, n1 - 2.0) \end{aligned}$$

Then some of the possible results of a recombination on successor P3 are:

Replace an entire condition-successor pair:

$$\begin{aligned} P3(n0, n1) : & \; \mathbf{n1 > 3.0 \rightarrow b(3.0)\ a(2.0)\ c(1.0)} \\ & \; n0 > 2.0 \rightarrow d(4.0)\ [\ P1(n1 - 1.0, n0/2.0)\ ] \end{aligned}$$

Replace just a successor:

$$\begin{aligned} P3(n0, n1) : & \; n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1)\ c(3.0) \\ & \; n0 > 2.0 \rightarrow \mathbf{P1(n1 - 1.0, n1 - 2.0)} \end{aligned}$$

Replace one block with another:

$$\begin{aligned} P3(n0, n1) : & \; n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1)\ c(3.0) \\ & \; n0 > 2.0 \rightarrow d(4.0)\ [\ \mathbf{b(3.0)\ a(2.0)}\ ] \end{aligned}$$

3 Evolved Creatures

To evolve moving creatures we set their fitness to be a function of the distance moved by the creature's center of mass. Stepping and rolling motions were encouraged by applying a penalty proportional to the distance covered by points that dragged along the ground. Robust locomotion strategies were

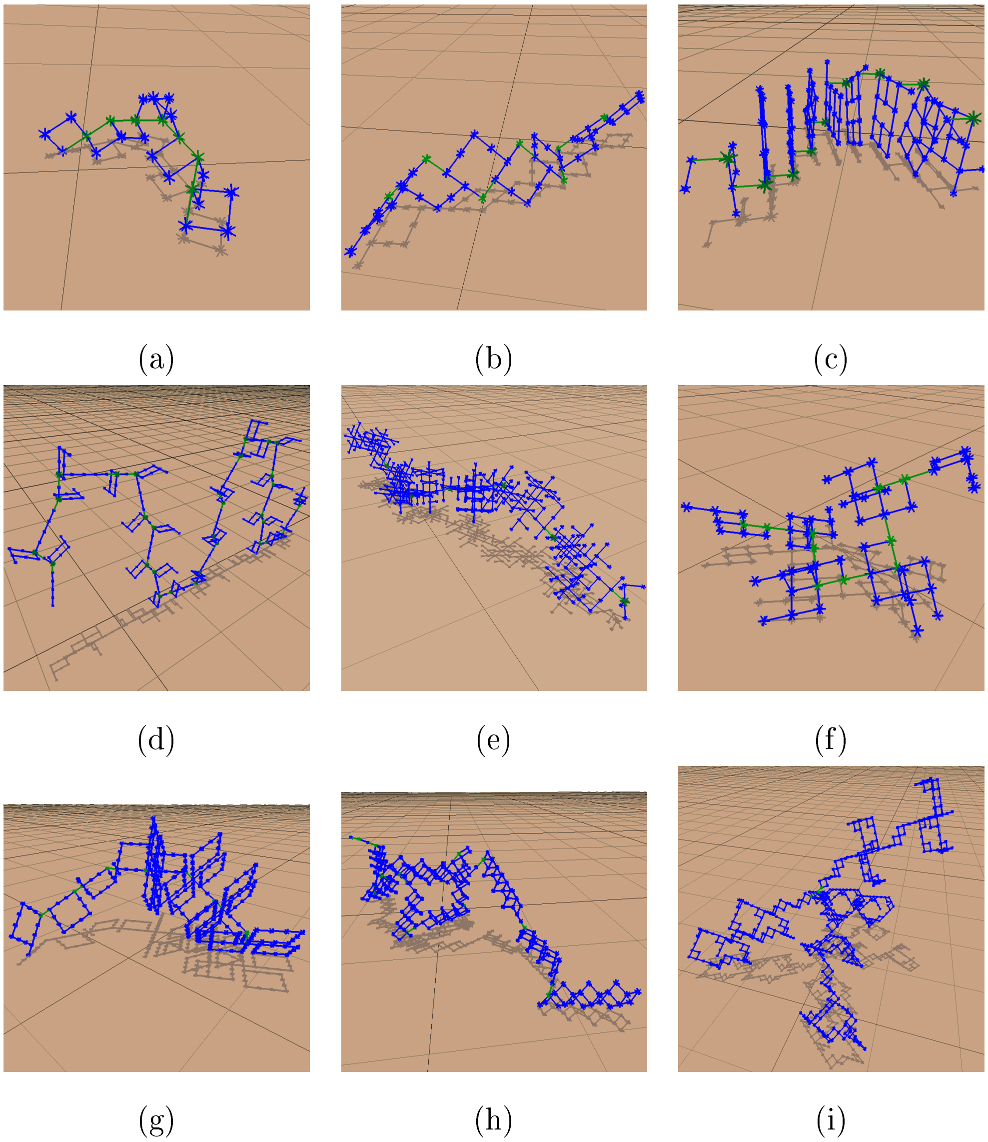


Fig. 2. A variety of evolved creatures: a, a rolling creature with 33 bars; b, a bi-connected rolling chain with 59 bars; c, a sequence of rolling rectangles with 169 bars; d, an undulating serpent with 339 bars; e, a 5-segmented inch-worm with 414 bars; f, a flipping creature with 99 bars; g, an asymmetric rolling creature with 306 bars; h, a coiling snake-like creature with 342 bars; and i, a four-legged walking creature with 629 bars.

encouraged by adding a random perturbation to joints not part of a cycle of bars. As this joint noise affects how the creature moves, three evaluation trials are made, each using different random noise, with the lowest score assigned as the creature's fitness.

The evolutionary algorithm was configured to run with a population of 100 individuals for a maximum of 500 generations. Experiments were run using from 10 to 20 production rules, 2 to 4 condition-successor pairs and 1 to 3 parameters for production rules.

Of the several dozen evolutionary runs made, approximately half produced interesting results. The most common form of movement for evolved creatures was to roll along sideways, as done by the chains in figures 2.a, 2.b, 2.c and 2.g. The creature in 2.d moves like an undulating sea-serpent and, in a similar way, the creature in figure 2.e moves like an inch-worm. Another common form of locomotion, similar to rolling, is the flipping of the creature in figure 2.f. Instead of performing a continuous rotation of its body, this creature repeatedly moves its center of mass outside its contact points and falls over. Two of the larger creatures that evolved are the one in figure 2.h, which moves by pushing a coil from front to back and then re-creating the coil at its front, and the creature in figure 2.i which uses four legs in an awkward walk.

4 Discussion

For this work we used parametric L-systems as a way to increase complexity over basic L-systems. Using parameters also has the advantage of allowing one production rule to be used to generate a class of objects, such as by using the parameter to specify the size of an attribute generated by the production rule or number of times to perform a loop. In this way parameters are analogous to the arguments of a function in a computer program and the evolution of an L-system becomes like the evolution of a computer program, as in genetic programming [17].

Two other types of L-systems that could have been used to achieve greater complexity than basic L-systems are stochastic L-systems and context sensitive L-systems [22]. Stochastic production rules have multiple successors, each with a probability that it will be used to replace the predecessor. The ability to use different successors to replace the same predecessor gives stochastic L-systems the ability to generate a variety of similar structures with the same set of rules, but because this is done non-deterministically this system is unsuitable for developing structures that need to be re-created the same each time. Another way in which variation can be applied to an L-system is through the addition of context. Context sensitive L-systems have conditions that examine the characters to the left and right of the character to be rewritten to determine which successor to replace it with. While this class of L-systems is deterministic, it is not as powerful a class of L-systems as are parametric L-systems [19] and not having parameters results in the inability to take advantage of parametric terminals, such as the oscillators whose parameter specifies the speed of oscillation.

In addition to the results presented here, this system can be modified to produce different styles of creatures or extended to other substrates. A different style of creatures can be produced simply by changing the amount of rotation performed by construction commands, such as the $60°$ system that was used for our 2D work [11], or different creature parts can be added, such as linear actuators to produce creatures of the style of [20]. Alternatively, this system can be used to generate designs for other substrates by changing the language of terminals and the construction module. Examples of L-systems evolved for different substrates are architectural floor plans [5], neural-networks [15,13], plants [14,24], and tables [12].

5 Conclusion

A system for creating virtual creatures was achieved by using evolutionary techniques to evolve parametric Lindenmayer systems. Using this system morphologies and controllers were evolved for moving creatures. The evolved creatures consist of an order of magnitude more parts and a higher degree of regularity than [23,16,20]. Already our creatures are capable of playing background characters with insect-level behaviors for animations. As we move from simple oscillator circuits to neural-controllers [13], which can sense and respond to the virtual environment, we expect to develop realistic artificial life-forms able to take a larger roll in future animations.

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Administration (DARPA) Grant, DASG60-99-1-0004. The authors would like to thank the members of the DEMO Lab: A. Bucci, E. DeJong, S. Ficici, P. Funes, S. Levy, H. Lipson, O. Melnik, E. Sklar, S. Viswanathan and R. Watson.

References

[1] H. Abelson and A. A. deSessa. *Turtle Geometry.* M.I.T. Press, 1982.

[2] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In R. K. Belew and L. B. Booker, editors, *Proc. of the Fourth Intl. Conf. on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1991.

[3] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies of an evolutionary design problem. In Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiel, and Smith, editors, *Genetic and Evolutionary Computation Conference*, pages 35–43, 1999.

[4] P. J. Bentley. *Generic Evolutionary Design of Solid Objects Using a Genetic Algorithm.* PhD thesis, University of Huddersfield, 1996.

[5] P. Coates, T. Broughton, and H. Jackson. Exploring three-dimensional design worlds using lindenmayer systems and genetic programming. In P. J. Bentley, editor, *Evolutionary Design by Computers*, 1999.

[6] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution.* Wiley Publishing, New York, 1966.

[7] P. Funes and J. Pollack. Computer evolution of buildable objects. In Phil Husbands and Inman Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life*, pages 358–367, Cambridge, MA, 1997. MIT Press.

[8] L. Gritz and J. K. Hahn. Genetic programming for articulated figure motion. *Journal of Visualization and Computer Animation*, 6(3):129–142, July 1995.

[9] R. Grzeszczuk and D. Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 63–70, 1995.

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[11] G. S. Hornby, H. Lipson, and J. B. Pollack. Evolution of generative design systems for modular physical robots. In *Intl. Conf. on Robotics and Automation*, pages 4146–4151, 2001.

[12] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Congress on Evolutionary Computation*, pages 600–607, 2001.

[13] G. S. Hornby and J. B. Pollack. Body-brain coevolution using L-systems as a generative encoding. In *Genetic and Evolutionary Computation Conference*, pages 868–875, 2001.

[14] C. Jacob. Evolution programs evolved. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature PPSN-IV*, Lecture Notes in Computer Science 1141, pages 42–51, Berlin, 1996. Springer-Verlag.

[15] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[16] M. Komosinski and A. Rotaru-Varga. From directed to open-ended evolution in a complex simulation model. In Bedau, McCaskill, Packard, and Rasmussen, editors, *Artificial Life 7*, pages 293–299, 2000.

[17] J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, Cambridge, Mass., 1992.

[18] A. Lindenmayer. Mathematical models for cellular interaction in development. parts I and II. *Journal of Theoretical Biology*, 18:280–299 and 300–315, 1968.

[19] A. Lindenmayer. Adding continuous components to L-Systems. In G. Rozenberg and A. Salomaa, editors, *L Systems*, Lecture Notes in Computer Science 15, pages 53–68. Springer-Verlag, 1974.

[20] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.

[21] J. T. Ngo and J. Marks. Spacetime constraints revisited. In *SIGGRAPH 93 Conference Proceedings*, pages 343–350, 1993. Annual Conference Series.

[22] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[23] K. Sims. Evolving Virtual Creatures. In *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 15–22, 1994.

[24] C. Traxler and M. Gervautz. Using genetic algorithms to improve the visual quality of fractal plants generated with csg-pl-systems. In *Proc. Fourth Intl. Conf. in Central Europe on Computer Graphics and Visualization 96*, 1996.

[25] M. van de Panne and E. Fiume. Sensor-actuator Networks. In *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 335–342, 1993.