Evolution of Vertex and Pixel Shaders

Marc Ebner, Markus Reinhardt and Jürgen Albert

Universität Würzburg, Lehrstuhl für Informatik II
Am Hubland, 97074 Würzburg, Germany
ebner@informatik.uni-wuerzburg.de
http://www2.informatik.uni-wuerzburg.de/staff/ebner/welcome.html

Abstract. In real-time rendering, objects are represented using polygons or triangles. Triangles are easy to render and graphics hardware is highly optimized for rendering of triangles. Initially, the shading computations were carried out by dedicated hardwired algorithms for each vertex and then interpolated by the rasterizer. Todays graphics hardware contains vertex and pixel shaders which can be reprogrammed by the user. Vertex and pixel shaders allow almost arbitrary computations per vertex respectively per pixel. We have developed a system to evolve such programs. The system runs on a variety of graphics hardware due to the use of NVIDIA's high level Cg shader language. Fitness of the shaders is determined by user interaction. Both fixed length and variable length genomes are supported. The system is highly customizable. Each individual consists of a series of meta commands. The resulting Cg program is translated into the low level commands which are required for the particular graphics hardware.

1 Motivation

In computer graphics three dimensional objects are usually represented using polygons. Polygons in turn can be broken down to triangles. A triangle is to computer graphics what the atom is to chemistry. Even curved objects such as spheres or cylinders are approximated with triangles. The surface nevertheless appears round due to special shading techniques. The advantage of using triangles is that the graphics pipeline can be highly optimized. A triangle consists of three vertices. Each vertex is assigned a number of attributes such as color, reflectance properties or a normal vector. Initially, graphics libraries used fixed algorithms to compute the color of a vertex using the assigned reflectance properties of the material it is supposed to represent [1, 6, 24]. After the color of the vertex is calculated, the polygon or triangle is filled by interpolating the colors computed for the vertices. This method is called Gouraud shading. In todays graphic hardware these shading algorithms are no longer fixed, they can be reprogrammed by the user. This is done using pixel and vertex shaders [3, 10]. A vertex shader is a small program which computes or modifies attributes such as position, normal vector, or reflectance properties. These attributes are interpolated to obtain the data for each pixel. A pixel shader is used to compute the color of each pixel from these attributes. Both, vertex and pixel shaders are

programs which can be evolved. Use of genetic programming [2, 7, 8] to evolve shaders was originally suggested by Kenton Musgrave [12].

We have developed a system which allows us to evolve pixel and vertex shaders by user interaction [17]. The system starts off with a number of randomly created pixel shaders and a fixed vertex shader or vice versa. The pixel and vertex shaders are applied to an object which is shown to the user. The user can then judge how good the pixel respectively vertex shader is and set its fitness value. Genetic operators are applied and new shaders are created. Again the shaders are presented to the user which has to rate the quality of the shaders. This process can continue for as long as the user wants. In the following we will first summarize some background material on vertex and pixel shaders. Then we describe our system and the experiments we have made.

2 Vertex and Pixel Shaders

Vertex and pixel shaders can be reprogrammed using a custom assembly language. A vertex shader receives its input, the attributes of a vertex, through a fixed number of registers. This input is read-only. A vertex shader processes four dimensional data. Each register contains four floating point numbers which map naturally to the three color bands red, green, and blue. The fourth component describes how transparent the object is. A set of output registers is used to store the modified attributes. Another set of registers can be used during the computation. A small amount of memory can also be accessed read-only.

A vertex shader program consists of a sequence of commands. Originally, a vertex shader could contain a maximum of 128 commands. The set of commands included standard arithmetic operators such as addition, subtraction, multiplication and computation of the scalar product between two vectors. Apart from the standard operators, some commands also addressed the special needs in computer graphics such as the computation of coefficients for ambient, diffuse and specular lighting or the computation of coefficients for light attenuation. Originally, there were no explicit flow control statements such as `if`, `for`, `while` or `goto`. However, it was possible to implement if-then-else operations within the simple instruction set given. Subsequently vertex shader instructions now also contain flow-control instructions to jump forward, loop a fixed number of times, and call subroutines [1].

A pixel shader is used to compute the color of every pixel of a fragment. It receives the interpolated components such as diffuse and specular light which was computed by the vertex shader as input. The vertex shader also has access to multiple textures and can combine the diffuse and specular components with this texture data. The registers of a pixel shader contain four values where the red, green, blue and alpha components of a color are stored. There are also a number of registers where temporary data may be stored and some address registers through which texture data can be accessed. Like the vertex shader, a pixel shader is a small program. The difference is that the commands of the pixel shader are tailored for texture access. It consists of two set of commands,

arithmetic operations and operations for texture addressing. No explicit flow control statements are included. Using a pixel shader it is also possible to change the depth of a pixel or even end further processing of a patch.

3 Evolution of Vertex and Pixel Shaders

Both vertex and pixel shaders are basically short sequences of commands which can be evolved. Evolution of shaders was originally proposed by Kenton Musgrave [12]. Loviscach and Meyer-Spradow [9, 11] evolved vertex and pixel shaders using a one-to-one fixed length representation between the genotype and the assembly language of the hardware. In contrast to the system by Loviscach and Meyer-Spradow, we evolve shaders using NVIDIA's high level Cg shader language. This allows us to define arbitrary computer architectures for which shaders may be evolved.

We have used linear genetic programming [13, 14] to evolve vertex and pixel shaders. Each individual consists of a sequence of numbers from the range $[0, 255]$. We work with both fixed and variable length individuals. The information stored in an individual is mapped into a program as shown in Figure 1. A reading head moves along the individual and parses byte after byte. The first number is treated as an opcode of a command. Depending on the type of command we either need none, one, two or more arguments. If no arguments are needed then we proceed with the next byte and map this value into another command. Otherwise we fetch the required number of arguments from the individual and map these bytes to the corresponding variables. We then proceed with the next byte. In order to perform the mapping from bytes to commands respectively variables, we have defined two tables. One table lists the set of commands, the other lists the set of variables. A modulo operation is used in both cases to map any value from the range $[0, 255]$ to a valid command respectively to a valid variable.

We do not use a particular graphics hardware as our target. Instead, we have chosen to use NVIDIA's Cg Toolkit [15] to perform the final mapping to the graphics hardware. This allows us to produce vertex and pixel shaders for all current and hopefully all future hardware. When mapping individuals to vertex or pixel shaders we create an output in the Cg language. The Cg language is a high level language similar to C. A vertex or pixel shader is constructed by first defining a wrapper. This wrapper is the same for all individuals. The wrapper consists of a header and a footer. A shader is created by taking the header, appending the commands as specified by the genotype, and finally appending the footer.

Our system is extremely versatile in that we can define arbitrary computer architectures using the table of commands and the table of variables. These tables are not stored internally but can be modified by the user. By modifying the tables we can vary the basic architecture of the programs to be evolved. We can define architectures with either a one-address instruction, a two-address instruction or even stack based architectures. It is also possible to create new

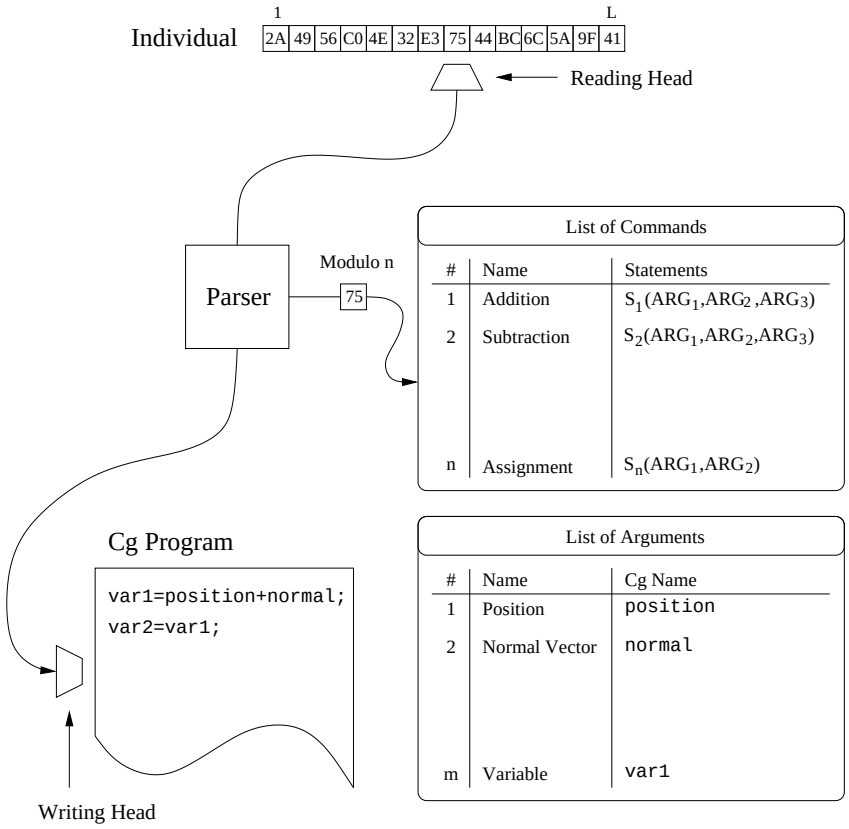

Fig. 1. An individual is mapped to a Cg program by moving a reading head along the bytes stored in the genotype. After reading the first byte we look up the corresponding set of statements and the number of required arguments in the list of arguments. The byte is fed through a modulo operation to access the table. Then the arguments are looked up in the second table. For each argument we parse another byte from the individual. After all arguments are available we append the statements to the Cg program. This process continues until all bytes of the individual have been parsed or we run out of bytes.

meta commands which are not part of the original language. Each entry of the command table can contain an arbitrary sequence of statements of the Cg language. The required arguments are accessed through escape sequences. When parsing the individual these escape sequences are mapped to the corresponding variables.

Currently, we use perceptual selection [20, 21] to evaluate the individuals. We work with small population sizes. All individuals are presented to the user for evaluation. The vertex respectively pixel shaders are applied to a three dimensional object for viewing. If the user likes what he sees, he sets the fitness of these

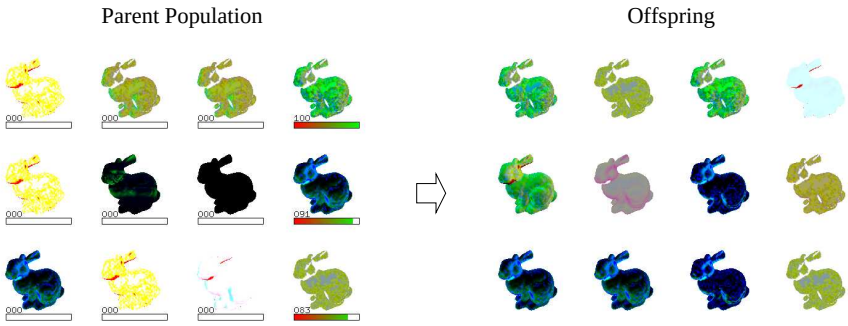

Fig. 2. A population of 12 individuals is shown on the left. The user manually sets the fitness of any individuals he likes. On the right we see the offspring population created from the three parent individuals.

individuals to a high value. Figure 2 shows one step of the evolutionary algorithm. On the left, we see a 4×3 matrix of individuals. The color bar below each individual is used to specify the fitness of an individual. Similar methods of interactive evaluation were used by Dawkins [4] when evolving his classic Biomorphs. Perceptual selection is also frequently used in evolutionary art systems [5, 18, 19, 22, 25]. Rowbottom gives a review of many of these systems [19].

When the user has finished evaluating the individuals (not all individuals have to be evaluated), the next generation of individuals is created. Both crossover and mutation operators are used. First we decide if a crossover is applied at all using a single crossover probability. Then we chose the actual crossover operator (one point or two point crossover) with uniform probability. After the two offspring are created, the individuals are mutated. The mutation probability is specified per byte. The type of mutation actually used is selected with uniform probability. Two types of mutation operations were defined for fixed length individuals: flip mutation and swap mutation. Flip mutation replaces one byte with a new value. Swap mutation swaps one byte with another byte of the same string. For variable length individuals, two additional mutation operators were used: insert and delete. The insert mutation operator inserts a new byte. The delete mutation operator deletes a byte from the individual.

When implementing evolutionary algorithms we usually want that parent and offspring are closely related. However our opcodes or meta commands require a variable number of arguments. If a single mutation were to change an opcode which requires a single argument to an opcode with a different number of arguments then this would have a large effect on the resulting Cg code. The entire code following the locus where the mutation occurs would be changed. Arguments would be interpreted as opcodes and vice versa. What we probably want is that a single mutation is able to either change the opcode or the argument into some other opcode or argument. Therefore we implemented a second parsing mode. Let k be the maximum number of arguments over all commands.

In this case, we fetch $k+1$ bytes (one byte for the opcode and k bytes for the arguments) from the individual. The two parsing modes are shown in Figure 3. Since we do not know which method of parsing the individuals leads to better results, we have implemented both methods. Again, the choice on how the individuals are evaluated, is left to the user.

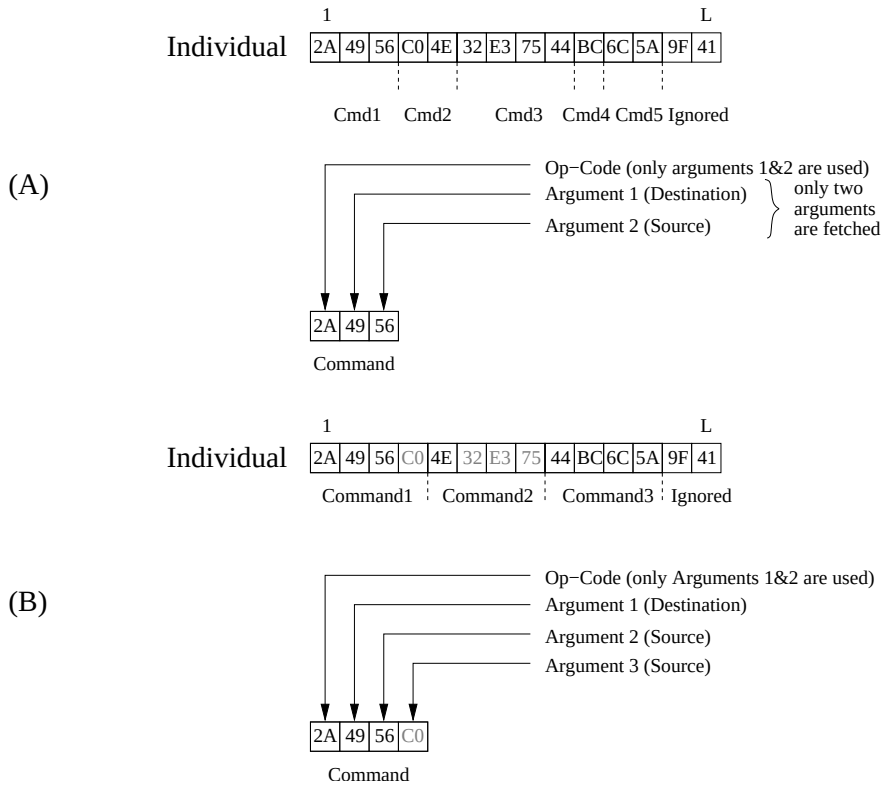

Fig. 3. Each individual consists of a sequence of commands. The commands themselves can either be considered to have variable length (A) or a fixed length (B). Using a fixed length for commands has the advantage that opcodes and arguments are always registered. Therefore, it is possible to change a single argument of an individual using a point mutation. A crossover exchanges opcodes with opcodes and arguments with arguments. This is not the case for (A). If variable length commands are used a single point mutation may have a large effect on the resulting Cg code.

4 Experiments

Figure 4 shows a collection of evolved vertex shaders. Each row shows the results for a single shader applied to four different shapes: a plane, a sphere, a torus and

Table 1. List of commands.

Operator	Operands	Result
Nop	0	No operation
Swap	0	Exchange registers 1 and 2
Noise	0	Noise function applied to register 1
Sin	0	Sine function applied to register 1
Normalize	0	Normalize register 1
Add	1	Add operand to register 1
Subtract	1	Add operand to register 1
Multiply	1	Multiply operand with register 1
Divide	1	Divide register 1 by operand, if operand is non-zero

Table 2. List of arguments.

Argument
Register 1
Register 2
Red vector
Green vector
Blue vector
Position of vertex
Normal vector
Eye vector
Light vector
Half vector between light and eye vector
Diffuse lighting (dot product between normal and light vectors)
Specular lighting (dot product between normal vector and half vector)
Diffuse and specular lighting
Eye to vertex vector
Animator (changing float value)

the Stanford bunny. The shaders were evolved during two runs lasting 130 and 185 generations. Both runs used a two register machine model. The contents of the two registers can be exchanged using a swap operation. Instructions include addition, subtraction, multiplication, protected division, and a sine function. A normalize function and the popular noise function [16] was also included. The set of meta commands is shown in Table 1. The list of arguments is shown in Table 2. Arguments include the two registers, three color vectors red, green and blue, the current position of the vertex, normal vector, eye vector, light vector as well as some pre-calculated values such as the half vector, diffuse and specular lighting. An animator (a float value which changes periodically) is also included. The animator can be used to create animated shaders. Output of register 1 is used as the color of the vertex. We have used a population size of 12. All shaders shown in Figure 4 were evolved using fixed length individuals of length 20. Mutation probability was set to $\frac{1}{20}$ which resulted in one mutation per offspring. Crossover

probability was set to 0.9. Roulette wheel selection was used to select offspring for breeding.

Although we were able to evolve some nice shaders we also noticed some limitations with the current approach. It is hard to evolve towards a particular target. For instance, it would be nice to be able to select two individuals and then obtain offspring which contain traits from both parents. If two individuals are selected, offspring may have interesting traits but may not have the intended look to them. I.e., if one selects a textured individual and another individual with a different color then the next generation will contain all types of individuals but not necessarily an individual with both the texture of one parent and the color of the second parent. This may be caused by a number of factors. First of all we are working with very small population sizes because fitness has to be determined by the user. Another cause may be the use of linear individuals. It may be that a tree based genetic representation is more amenable to evolution in this case. Rowbottom [19] noted that most evolutionary art systems have a certain signature to them. This also seems to be the case here. The evolved individuals seem to be largely a function of the type of commands and arguments used.

5 Conclusion

Vertex and pixel shaders are an exciting concept of computer graphics. We have developed a system to evolve vertex and pixel shaders via user interaction. Individuals are interpreted as linear sequences of commands which are translated into a high level computer graphics language. Individuals are applied to four different objects and presented to the user who then decides which individuals get to produce offspring.

Our system is highly customizable. With this system it is possible to define virtual intermediate architectures. At present, it is not known which architecture is best suited to evolve vertex and pixel shaders. Our initial experiments focused on the evolution of linear programs. It would be interesting to see how tree based genetic programming compares to linear genetic programming for the evolution of vertex and pixel shaders.

Another possible extension would be the automatic evolution of shaders for animated effects. One could take a short sequence of a movie taken with a digital camera and then evolve shaders which mimic the effect seen in the video. Other than evolving vertex and pixel shaders for computer graphics the concept may also be of interest to other researches who want to speed up their genetic programming experiments. It may be possible to use vertex or pixel shaders in other areas such as evolution of classifiers.

6 Acknowledgments

Our system uses the GAlib genetic package version 2.4, written by Matthew Wall at the Massachusetts Institute of Technology [23].

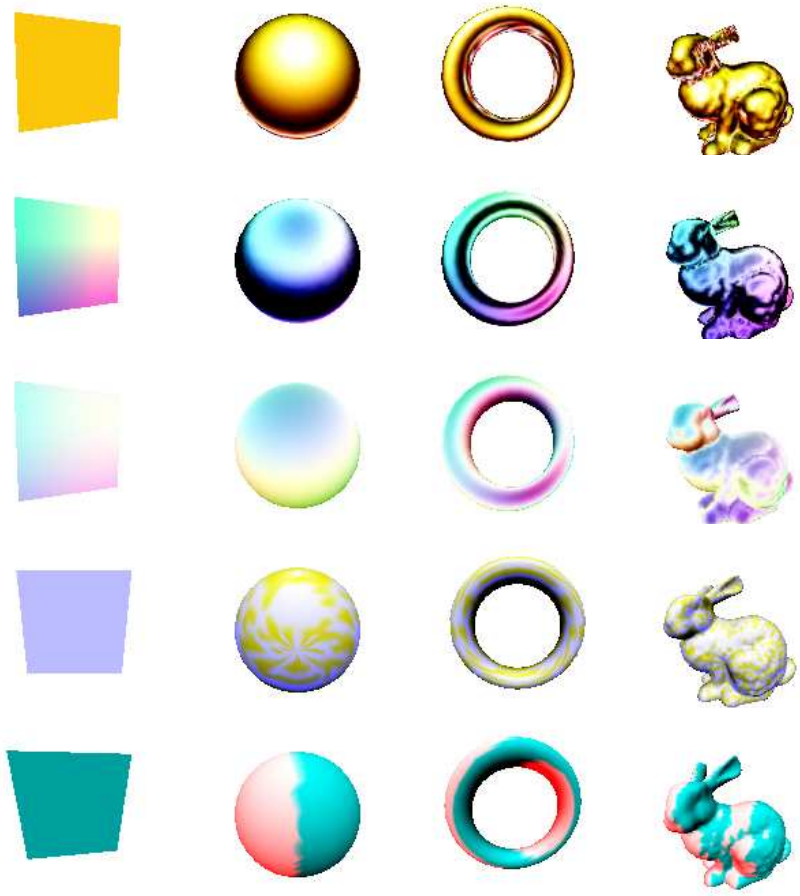

Fig. 4. A collection of evolved vertex shaders. Each row shows the results for a single vertex shader. A vertex shader is applied to four different objects: a plane, a sphere, a torus and the Stanford bunny.

References

1. T. Akenine-Möller and E. Haines. *Real-Time Rendering*. A K Peters, Natick, MA, 2nd ed., 2002.
2. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming - An Introduction: On The Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.
3. NVIDIA Corporation. Nvidia nfinitefx engine: Programmable vertex shaders.
4. R. Dawkins. *The Blind Watchmaker*. W. W. Norton & Company, New York, 1996.
5. A. E. Eiben, R. Nabuurs, and I. Booij. The Escher evolver: Evolution to the people. In P. J. Bentley and D. W. Corne, eds., *Creative Evolutionary Systems*, pp. 425–439. Morgan Kaufmann Publishers, 2001.

6. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice. 2nd Ed. in C.* Addison-Wesley Publishing Company, Reading, MA, 1996.
7. J. R. Koza. *Genetic Programming. On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, MA, 1992.
8. J. R. Koza. *Genetic Programming II. Automatic Discovery of Reusable Programs.* The MIT Press, Cambridge, MA, 1994.
9. J. Loviscach and J. Meyer-Spradow. Genetic programming of vertex shaders. In *Proc. of EuroMedia 2003*, pp. 29–31, 2003.
10. C. Maughan and M. Wloka. Vertex shader introduction. Technical report, NVIDIA Corporation, 2001.
11. J. Meyer-Spradow and J. Loviscach. Evolutionary design of BRDFs. In M. Chover, H. Hagen, and D. Tost, eds., *Eurographics 2003 Short Paper Proceedings*, pp. 301–306, 2003.
12. F. Kenton Musgrave. Genetic textures. In D. S. Ebert, F. Kenton Musgrave, D. Peachey, K. Perlin, and S. Worley, editors, *Texturing and Modeling: A Procedural Approach. 2nd Ed.*, pp. 373–385, Cambridge, 1998. AP Professional.
13. P. Nordin. A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr., ed., *Advances in Genetic Programming*, pp. 311–331, Cambridge, MA, 1994. The MIT Press.
14. P. Nordin and W. Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behaviour*, 5(2):107–140, 1997.
15. NVIDIA. Cg toolkit. user's manual. a developer's guide to programmable graphics. Technical report, NVIDIA Corporation, Santa Clara, CA, 2002.
16. K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH '89 Conference Proceedings, Computer Graphics, Boston, MA*, 23(3):253–262, 1989.
17. M. Reinhardt. *Evolution von Pixel- und Vertex-Shader Programmen.* Projektpraktikum, Universität Würzburg, Institut für Informatik, Lehrstuhl für Informatik II, July 2004.
18. S. Rooke. Eons of genetically evolved algorithmic images. In P. J. Bentley and D. W. Corne, eds., *Creative Evolutionary Systems*, pp. 339–365. Morgan Kaufmann Publishers, 2001.
19. A. Rowbottom. Evolutionary art and form. In P. J. Bentley, ed., *Evolutionary Design by Computers*, pp. 261–277, San Francisco, 1999. Morgan Kaufmann.
20. K. Sims. Artificial evolution for computer graphics. *Computer Graphics*, 25(4):319–328, 1991.
21. K. Sims. Interactive evolution of dynamical systems. In F. J. Varela and P. Bourgine, eds., *Toward a practice of autonomous systems: Proc. of the 1st Europ. Conf. on Artificial Life*, pp. 171–178, Cambridge, MA, 1992. The MIT Press.
22. S. Todd and W. Latham. The mutation and growth of art by computers. In P. J. Bentley, ed., *Evolutionary Design by Computers*, pp. 221–250, San Francisco, 1999. Morgan Kaufmann.
23. M. Wall. *GAlib: A C++ Library of Genetic Algorithm Components, Version 2.4.* Mechanical Engineering Department, Massachusetts Institute of Technology, 1996.
24. A. Watt. *3D Computer Graphics.* Addison-Wesley, Harlow, England, 2000.
25. M. Witbrock and S. Neil-Reilly. Evolving genetic art. In P. J. Bentley, ed., *Evolutionary Design by Computers*, pp. 251–259, San Francisco, 1999. Morgan Kaufmann.