Evolutionary Design of BRDFs

J. Meyer-Spradow

Fachbereich Informatik, FG Graphische Datenverarbeitung, Universität Hannover, Hannover, Germany

J. Loviscach

Fachbereich Elektrotechnik und Informatik, Hochschule Bremen, Bremen, Germany

Abstract

The look of a non-transparent material is determined by its bidirectional reflection distribution function (BRDF). To design 3-D objects for example for games or animation films thus includes to design BRDFs. However, as functions defined on a four-dimensional domain, these form a vast space that is very difficult to explore interactively. Typically, the infinite number of degrees of freedom is reduced to a tractable handful of parameters by introducing simplified physical models or heuristic approximations such as Phong's. As the complexity of such approaches increases, they become difficult to master for a human operator. Even if many parameters are made accessible, an infinite variety of useful and/or interesting BRDFs remains hidden and inaccessible. We therefore propose a method of constructing BRDFs through genetic programming with a human operator making choices based on his or her preferences. With the pixel shader programmability of modern graphics cards this can be performed in real time.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

For adjusting the materials of the objects to be rendered, typical 3-D design software offers a basic choice of shaders such as Phong or Blinn. As they contain only a small number of control parameters they are quite easy to understand, but make the user stuck with established lighting models, often creating a clean or stereotyped look.

While some advanced shaders such as bhodiNut for Maxon Cinema 4D or the water shader of Alias|Wavefront Maya™ achieve more complex results, this comes at the cost of either having to adjust dozens of parameters or being limited to one special kind of material modeled with large precision. In summary this means that today's software hardly allows exploring the space of materials.

A similar problem arises already with two-dimensional texture images. However, here software such as Corel™ (Ex-MetaCreations) KPT Texture Explorer® has long been used to design textures by evolution. The user looks at a small set of genetic mutations and chooses from them a single base genotype for the next round of mutations. This corresponds to genetic programming; however, the fitness of an individual (in this case a texture) is not determined algorithmically but by the user judging aesthetically. This allows to steer the search through the space of textures—a space with virtually infinite degrees of freedom.

Thanks to the programmability of modern graphics hardware, a similar solution becomes feasible now for BRDFs. Our prototype implementation (see Fig. 1) loads an polygonal 3-D object including texture coordinates from an standard .OBJ file. The generated materials (i. e., BRDFs) are displayed in real time on this model. Following the established user interface of KPT Texture Explorer®, the software shows in 20 smaller frames around a large central display genetic variations (mutations) of the material shown in large. A point light source automatically circles around the displayed objects; for clarity it is marked by a white dot.

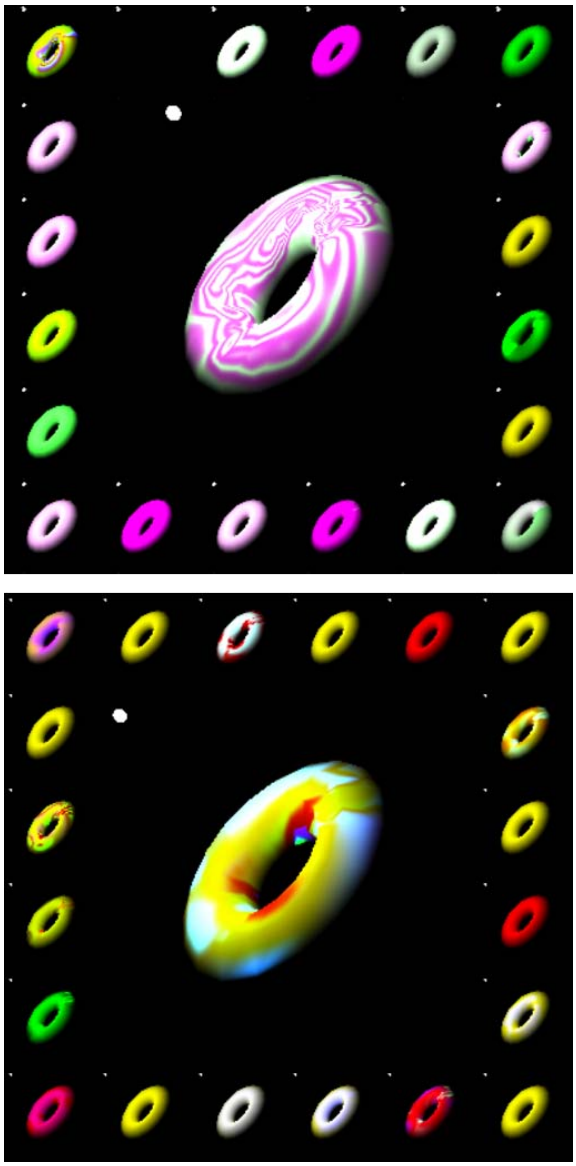

Figure 1: *Genetically programmed BRDFs composed of approximately 100 shader instructions are displayed in real time under a moving light source. The user steers the genetic evolution by aesthetic selection.*

Clicking with the left mouse button on any of the frames displays the respective material in the center and produces the variations shown around of it.

As an extension of the well-known user interface of KPT Texture Explorer®, one can click with the right mouse button onto a frame and then with the left mouse button onto another to produce genetic mixtures (recombinations) of both. With help of the middle mouse button, the 3-D object can be rotated; using the CTRL key together with the middle mouse button, it can be scaled. The implementation prototype saves and loads generated materials in a proprietary format. The amount of randomness used for mutation and recombination and the rotation speed of the light source can be controlled by the user. Additionally, one can restrict the generated materials to be homogeneous (BRDF not depending on the position on the surface) and/or isotropic (unchanged by rotations about the surface normal).

The software has been implemented in C++ using OpenGL™. Using BRDFs consisting of 100 assembler instructions in the genetically programmed part of the pixel shader, the prototype achieves a speed of approximately 25 frames per second under Microsoft® Windows XP® on a PC equipped with an Intel® Pentium® 4 running at 2.4 GHz and a graphics card nVidia® GeForce™ FX 5800.

The main contributions of this work are the methods employed to generate and to genetically program BRDFs via pixel shaders. The paper is organized as follows: In Section 2 we outline related work in the field of evolutionary design and parametrization of BRDFs. Section 3 introduces our representation of BRDFs by programs. How these programs are implement by pixel shaders and how genetic operations can be defined on them is described in Section 4. We discuss our results and give an outlook on future work in Section 5.

2. Related Work

Sims[20] proposed to employ genetic programming[10] not with automatic evaluation of the fitness but with aesthetic selection by a human. This idea has found many applications[2, 3] to art and to graphic design, even in such areas as the creation of music and 3-D characters. An similar approach[14] aims at designing video textures in real time, employing the vertex shader programmability already found in the last but one generation of graphics cards. Abraham's Genshade[7] evolves RenderMan shaders either with or without human supervision. Some recent works such as Gentropy by Wiens and Ross[22] use genetic programming to generate images, but evaluate the results automatically. In the same spirit, Hewgill and Ross[6] evolve 3-D textures, feeding position coordinates, normal vector and surface gradient into LISP programs.

Genetic operations acting directly on machine code typically need strong restrictions[5] and/or measures against errors[11]. However, the pixel shaders of the class of 3-D chips employed here only consist of linear code, without loops or branches. No error conditions are possible if only syntactically correct opcodes are given. Therefore, such programs can easily be mutated and recombined as instruction lists. Their overall structure resembles Cartesian Genetic Programming[17].

The seminal work of Phong[18] may be viewed as a forerunner to modern research into compact representations and

handy simulations of reflective properties of materials. For instance, Ashikhmin[1] and co-workers propose a widely parameterizable microfacet model.

Another strand of research started with Fournier's[4] singluar value decomposition of directional dependence. Kautz[8], McCool[16] and co-workers use similar factorizations to parametrize BRDFs for real-time rendering. Latta and Kolb[12] extend this approach to include global lighting effects. Kautz[9] and coworkers as well as Ramamoorthi and Hanrahan[19] use a small number of spherical harmonics to efficiently convolve a lighting distribution with the BRDFs. In a different approach, Malzbender[15] and co-workers use biquadratic functions to approximate direction-dependent terms in an extended version of Phong's lighting model.

3. Parameters for BRDFs

The BRDF f of a material at a single wavelength and a single point on a surface maps the incoming distribution L_I of radiance to the outgoing L_O via

$$L_\mathrm{O}(\omega_\mathrm{O}) = \int_\Omega f(\omega_\mathrm{O},\omega_\mathrm{I}) L_\mathrm{I}(\omega_\mathrm{I}) \cos(\theta_\mathrm{I})\, d\sigma(\omega_\mathrm{I})$$

where the integral runs over the half sphere Ω pointing outside and where θ_I measures the angle between the surface normal and the incoming direction (see Fig. 2). For a point-like light source at direction ω_I this expression simplifies to

$$L_\mathrm{O}(\omega_\mathrm{O}) = f(\omega_\mathrm{O},\omega_\mathrm{I}) I (\cos(\theta_\mathrm{I}))_+$$

with I denoting the irradiance due to the light source and $(x)_+ := x$ if $x > 0$ and $(x)_+ := 0$ otherwise. A physically realizable BRDF can only have non-negative values and must be reciprocal, which means

$$f(\omega_\mathrm{O},\omega_\mathrm{I}) = f(\omega_\mathrm{I},\omega_\mathrm{O}) \geq 0$$

for all incoming and outgoing directions. Additionally, it must not lead to energy production:

$$1 \geq \int_\Omega f(\omega_\mathrm{O},\omega_\mathrm{I}) \cos(\theta_\mathrm{O})\, d\sigma(\omega_\mathrm{O}) \qquad (1)$$

for all incoming directions ω_I.

Our objective is to convert an arbitrary pixel shader g to a BRDF. Its input consists of data about ingoing and outgoing directions (more on that below); its output is a triple of floating point values (r,s,t):

$$(r,s,t) = g(\omega_\mathrm{I},\omega_\mathrm{O})$$

The pixel shader constructed will be manifestly reciprocal:

$$g(\omega_\mathrm{I},\omega_\mathrm{O}) = g(\omega_\mathrm{O},\omega_\mathrm{I})$$

for all directions. To achieve non-negativity, we form

$$h(\omega_\mathrm{I},\omega_\mathrm{O}) := g(\omega_\mathrm{I},\omega_\mathrm{O})^2,$$

where $\cdot^2$ denotes component-wise squaring.

As input to the pixel shader we have to provide data about ingoing and outgoing directions. This is mainly calculated in an additional initial stage prefixed to the pixel shader. We use the following scalar quantities:

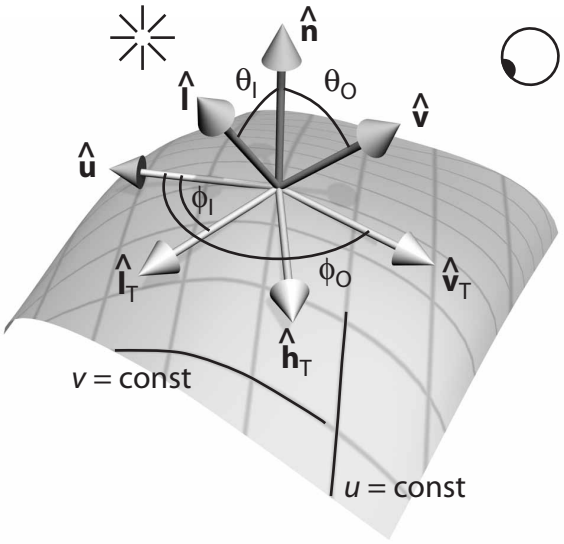

Figure 2: *Geometric parameters of the BRDF.*

$$\begin{aligned}
p_1 &:= \cos(\theta_\mathrm{O}) + \cos(\theta_\mathrm{I}) \\
p_2 &:= \cos(\theta_\mathrm{O}) - \cos(\theta_\mathrm{I}) \\
p_3 &:= \cos(\phi_\mathrm{O} - \phi_\mathrm{I}) \\
p_4 &:= \sin(\phi_\mathrm{O} - \phi_\mathrm{I}) \\
p_5 &:= \cos(\phi_\mathrm{O} + \phi_\mathrm{I}) \\
p_6 &:= \sin(\phi_\mathrm{O} + \phi_\mathrm{I}),
\end{aligned}$$

where we measure all azimuthal angles ϕ relatively to the u-direction of the texture coordinates.

These six parameters are sufficient to specify ingoing and outgoing directions for the BRDF. Any isotropic BRDF can be evaluated completely using only p_1 to p_4. There is some redundancy in these parameters, for instance $p_3^2 + p_4^2 = 1$. However, this helps to avoid artificial discontinuities in the basic parameters such as at $360^\circ \to 0^\circ$ in ϕ_O. In order that the generated BRDF behaves continuously, we only use continuously varying parameters as inputs.

In addition, this symmetric choice of parameters makes it simple to construct an arbitrary *reciprocal* BRDF: As one switches the roles of ingoing and outgoing directions, the parameters p_1, p_3, p_5, p_6 stay constant and p_2, p_4 only change sign. Therefore, reciprocity can be guaranteed if the BRDF remains unchanged if both p_2 and p_4 are multiplied by -1. But any function f that for all x and y fulfills $f(x,y) = f(-x,-y)$ must only depend on the complex square $(x+iy)^2$ and can thus be written $f(x,y) = g(x^2-y^2, 2xy)$ with an appropriate function g. We can thus achieve manifestly reci-

procity by using as inputs for the main part of the pixel shader: p_1, p_3, p_5, p_6, $p_2^2 - p_4^2$, $2p_2p_4$, u, and v. Vice versa, any reciprocal BRDF can be generated from these parameters. To optionally constrain the generated BRDFs to the case of isotropic and/or homogeneous materials, the parameters p_5, p_6 or u, v, respectively, can be ignored.

Given the geometry of the 3-D scene, the parameters p_1, …, p_6 are computed as follows: Per vertex the following vectors are determined in world coordinates and sent to the graphics card: normal direction $\hat{\mathbf{n}}$, direction to light $\hat{\mathbf{l}}$, direction to viewer $\hat{\mathbf{v}}$, u-direction of texture $\hat{\mathbf{u}}$, and optionally texture coordinates (u,v) for position-dependent BRDFs. (Here and in the following, boldface letters denote vectors and hats denote normalized vectors.)

The vector $\hat{\mathbf{u}}$ pointing in the local direction of the texture coordinate u is determined per vertex as follows: Consider the vertices $\mathbf{x}_k$, $k = 1,2,\ldots$, connected via a single edge to a given vertex $\mathbf{x}_0$. Let their texture coordinates be (u_k, v_k) and (u_0, v_0), respectively. Then we seek (unnormalized) vectors $\mathbf{u}$ and $\mathbf{b}$ in the local tangent plane ($\mathbf{u}\cdot\hat{\mathbf{n}} = 0$ and $\mathbf{b}\cdot\hat{\mathbf{n}} = 0$) yielding an optimal linear approximation in the sense that

$$\sum_k \left| \mathbf{x}_0 + (u_k - u_0)\mathbf{u} + (v_k - v_0)\mathbf{b} - \mathbf{x}_k \right|^2$$

attains a minimum. Through variation with Lagrange multipliers, $\mathbf{u}$ can be found as

$$\frac{\begin{vmatrix} \sum_k (u_k-u_0)\pi_T(\mathbf{x}_k-\mathbf{x}_0) & \sum_k (u_k-u_0)(v_k-v_0) \\ \sum_k (v_k-v_0)\pi_T(\mathbf{x}_k-\mathbf{x}_0) & \sum_k (v_k-v_0)^2 \end{vmatrix}}{\begin{vmatrix} \sum_k (u_k-u_0)^2 & \sum_k (u_k-u_0)(v_k-v_0) \\ \sum_k (u_k-u_0)(v_k-v_0) & \sum_k (v_k-v_0)^2 \end{vmatrix}},$$

where $\pi_T(\mathbf{x}) := \mathbf{x} - (\mathbf{x}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$ denotes the projection onto the tangent plane and where the determinant in the numerator has to be computed separately for each component of $\pi_T(\mathbf{x}_k - \mathbf{x}_0)$ yielding the corresponding component of $\mathbf{u}$. From normalization of $\mathbf{u}$ results $\hat{\mathbf{u}}$. The other vector $\mathbf{b}$ is neither used nor computed.

The graphics card interpolates the vectors $\hat{\mathbf{n}}$, $\hat{\mathbf{l}}$, $\hat{\mathbf{v}}$, $\hat{\mathbf{u}}$, and the scalars (u,v), which are given per vertex, to form corresponding vectors and scalars per pixel. In the initial phase of the pixel shader we compute a set of inputs for the main part of the pixel shader from these per-pixel values. While the texture coordinates (u,v) can be used directly, the vector quantities are no longer normalized after interpolation. Therefore, we divide them by their length to form pixel-wise vectors $\hat{\mathbf{n}}$, $\hat{\mathbf{l}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{u}}$.

As auxiliary unit vectors in the tangent plane we introduce $\hat{\mathbf{l}}_T := \widehat{\pi_T(\hat{\mathbf{l}})}$ and $\hat{\mathbf{v}}_T := \widehat{\pi_T(\hat{\mathbf{v}})}$. Now the parameters $p_1, \ldots, p_4$ can be computed:

$$\begin{aligned} p_1 &= \cos(\theta_O) + \cos(\theta_I) = (\hat{\mathbf{v}} + \hat{\mathbf{l}})\cdot\hat{\mathbf{n}} \\ p_2 &= \cos(\theta_O) - \cos(\theta_I) = (\hat{\mathbf{v}} - \hat{\mathbf{l}})\cdot\hat{\mathbf{n}} \\ p_3 &= \cos(\phi_O - \phi_I) = \hat{\mathbf{v}}_T \cdot \hat{\mathbf{l}}_T \\ p_4 &= \sin(\phi_O - \phi_I) = \hat{\mathbf{v}}_T \cdot (\hat{\mathbf{n}} \wedge \hat{\mathbf{l}}_T) \end{aligned}$$

We define the tangent half vector by $\hat{\mathbf{h}}_T := \left(\hat{\mathbf{v}}_T + \hat{\mathbf{l}}_T\right)^\wedge$. The vectors $\hat{\mathbf{v}}_T$ and $\hat{\mathbf{l}}_T$ possess azimuthal angles of ϕ_I and $\phi_O \in [0^\circ, 360^\circ)$ so that their normalized average $\hat{\mathbf{h}}_T$ has an azimuthal angle of $(\phi_I + \phi_O)/2$ if $|\phi_I - \phi_O| < 180^\circ$ and $(\phi_I + \phi_O)/2 \pm 180^\circ$ else. Noting that an offset of 180° cancels from the following equations, we can compute

$$\begin{aligned} p_5 &= \cos(\phi_O + \phi_I) \\ &= \cos((\phi_O + \phi_I)/2)^2 - \sin((\phi_O + \phi_I)/2)^2 \\ &= \left(\hat{\mathbf{h}}_T \cdot \hat{\mathbf{u}}\right)^2 - \left(\hat{\mathbf{h}}_T \cdot (\hat{\mathbf{n}} \wedge \hat{\mathbf{u}})\right)^2 \end{aligned}$$

and

$$\begin{aligned} p_6 &= \sin(\phi_O + \phi_I) \\ &= 2\cos((\phi_O + \phi_I)/2)\sin((\phi_O + \phi_I)/2) \\ &= 2\left(\hat{\mathbf{h}}_T \cdot \hat{\mathbf{u}}\right)\left(\hat{\mathbf{h}}_T \cdot (\hat{\mathbf{n}} \wedge \hat{\mathbf{u}})\right). \end{aligned}$$

4. Pixel Shader BRDFs and Genetic Programming

Via OpenGL™ extensions, the nVidia® GeForce™ FX 5800 offers per-pixel programmability through fragment programs (colloquially called pixel shaders). These are assembler programs called for every pixel fragment produced; their intended uses are complex lighting and texture computations. On the GeForce™ FX, a fragment program comprises up to 1024 instructions acting on registers storing four-component vectors. A "color fragment program", which is the type of pixel shader employed in this work, receives its input via twelve read-only attribute registers and writes its results into one or both of two result registers (color, depth).

For use as workspace, the chip offers a set of temporary registers, which are automatically initialized to zero. The number of temporary registers depends on the number of output registers used and the precision chosen: 16 bit or 32 bit floating point. Using only the color output register but not depth output, up to 63 temporary registers of 16 bit width can be addressed. In our experiments, the 16 bit operations turned out to be accurate enough and considerably faster than 32 bit operations.

Constant parameters can be embedded literally into assembler code or be written into 128 "local parameter registers" which are read-only from the pixel shader. We elected to use the latter ones in order to lessen the workload of the graphics driver: All programs are handed to the graphics driver as ASCII strings for assembling. There is no documented way of directly generating binary code.

While a typical job of pixel shaders consists of texture lookup, in our framework this is not necessary, so none of the texture lookup instructions are used in our prototype. For the

generated BRDFs to look plausible, we only employ instructions acting mathematically differentiably, see Table 1. We leave out instructions such as MAX (maximum), SGE (set on greater or equal) and RCP (reciprocal) because of the bends or discontinuities they can generate. The graphics chip offers to skip or execute instructions according to condition flags; for similar reasons, we make no use of this feature. On the fly, operands can be inverted or converted to absolute value and their components may be permuted at will ("swizzling"). Likewise, a result can be masked so that only some of the components of the targeted register are written into. To ensure mathematical differentiability, we use inversion, swizzling, and write masking, but not absolute value.

Opcode	Description
ADD	add two vectors
COS	cosine of a scalar
DP3	dot product using three components
DP4	dot product using four components
DST	auxiliary operation for light attenuation
EX2	the number 2 raised to a power given by a scalar
LRP	linear interpolation of two vectors
MAD	multiply and add vectors
MOV	copy vector to another register
MUL	multiply two vectors component-wise
SIN	sine of a scalar
SUB	subtract one vector from another
X2D	affine 2-D transform using two vectors

Table 1: *The instructions used in the genetically evolved part of the pixel shader.*

In summary, a pixel shader is a sequence of assembler instructions acting on certain registers and using fixed values stored in constant parameter registers. To program pixel shaders genetically, we encode an instruction by an opcode number, numbers for result and operand registers, and bit patterns for swizzling, inverting, and masking. Only one limitation of the hardware has to be taken into account: Each single instruction must not access more than one attribute or parameter register.

In the binary representation an instruction can be mutated for example by changing the opcode number. A program can be mutated by mutating randomly selected instructions in it. Using only programs of fixed length, it becomes simple to model genetic recombination: Just take two programs side by side and exchange random blocks at corresponding positions. For an example of pixel shader code generated by these operations acting together see Table 2.

To achieve a rate of 25 frames per second, we use pixel shaders whose genetic part is composed of 100 instructions. 40 additional pixel shader instructions are needed to prepare the parameters and compute the final color. We use no other temporary registers than those in which the parameters are prepared according to Section 3. If one uses more temporaries, the visual complexity drops because too few intermediate results are used in later steps.

ADDH	H5.xyz, -H1.yzyy, -H6.zyyx;
SUBH	H4.xw, -H6.xyxy, H2.xxxy;
COSH	H5.xy, H4.w;
MADH	H6.xy, -H1.zzxx, H5.xzzz, p[10].xxyy;
MOVH	H6.zw, -H3.xxxx;
DP4H	H5.xw, -p[35].xzzz, H6.yyxy;
COSH	H1.xw, -H5.x;
SINH	H3.xyw, p[35].z;
ADDH	H6.xyw, H1.yyyy, -H2.yyzz;
EX2H	H0.xyz, -H2.z;
EX2H	H5.xz, -H6.w;
MOVH	H3.yz, -H6.yzxy;
SINH	H0.yzw, p[11].z;
ADDH	H0.z, H1.zyyx, -H3.yyzx;
COSH	H4.xyz, H5.y;
MADH	H6.xyw, p[5].yxyx, -H2.yxzz, -H3.xzxx;
DSTH	H5.xyw, H4.xyyy, -H4.yyzy;
LRPH	H4.xyw, H6.zyyx, -H3.xzzy, H1.xyxy;
EX2H	H3.yz, -H5.z;
DP4H	H4.xz, p[33].yzxz, H0.zyyz;
MULH	H0.xw, -H0.yxxz, H4.xyyz;
SUBH	H5.xyz, -H0.zzyx, H5.zyyz;
DSTH	H0.xw, -H5.yyyy, p[5].zzzx;
DP3H	H0.w, p[10].yzzx, H3.xxzz;

Table 2: *Excerpt from a genetically generated pixel shader. The "H" designates a precision of 16 bit.*

Genetic Programming tends to bloat programs by computationally irrelevant code[21]. This may protect other code against the destructive actions of genetic operations. However, code bloat takes up valuable rendering time. Therefore, we use an optimization already implemented for vertex shaders[14]: Before converting the instruction sequence to an ASCII string to be sent to the graphics driver, a special routine hides all obviously superfluous instructions, i. e., such ones that only write into register components not used at a later stage. For programs consisting of 100 genetically evolved instructions we typically see 30 to 60 instructions being discarded this way. The speedup in rendering is nearly proportional to this figure.

5. Results, Outlook

We have designed and implemented a tool for interactive design of BRDFs via genetic programming of pixel shaders. Using the computing power of modern graphics cards, the system delivers even highly complex results in real time.

One may optimize the image quality using prefiltering to suppress aliasing artifacts and using gamma-curve predistortion to display the colors physically correct. To limit the outgoing energy to physically possible values according to Eqn. (1) did not prove to be visually important. However, one may simply include this constraint by computing

the maximum value of Eqn. (1) and dividing the generated BRDF by it. The maximum value can be approximated by repeatedly rendering a half sphere illuminated by a directional light source into an offscreen buffer.

In order to use the evolutionary designed BRDFs in standard 3-D animation and rendering software, we plan to implement an export feature. This will translate the pixel shaders into Cg or other shading languages which can be read by appropriate plug-ins. To facilitate such a scheme, each of the the computational steps done per vertex has to be implemented as an appropriate vertex shader or to be done in advance with its result stored as part of the model data. In particular, the vector $\hat{u}$ has to be part of the model because a vertex shader cannot process different vertices at once.

Our system already addresses spatially dependent BRDFs, i. e., BTFs (bidirectional texture functions). A realistic bump mapping could be achieved by generating meso-scale structures[13]. An automatic, not manual evolution may be guided by given BRDFs or even—as an "inverse rendering"—by given images. This mode could work similar as existing solutions searching for algorithms producing given 2-D textures[6 14].

References

1. M. Ashikhmin, S. Premože, and P. Shirley. A Microfacet-Based BRDF Generator. *ACM Computer Graphics (Proc. of SIGGRAPH 2000)*, pp. 65–74, 2000. 3
2. P. Bentley (ed.). *Evolutionary Design by Computers.* Morgan Kaufmann, 1999. 2
3. P. Bentley and D. W. Corne (ed.). *Creative Evolutionary Systems.* Morgan Kaufmann, 2002. 2
4. A. Fournier. Separating Radiosity Functions for Linear Radiosity. *Rendering Techniques '95 (Eurographics Workshop on Rendering)*, pp. 383–392, 1995 3
5. B. Harvey, J. Foster, and D. Frincke. Towards Byte Code Genetic Programming. *Proc. of the Genetic and Evolutionary Computation Conf. (Orlando)*, pp. 1234–1241, 1999. 2
6. A. Hewgill and B. J. Ross. *Procedural 3D Texture Synthesis Using Genetic Programming.* Technical Report # CS-03-06, Brock University, Dept. of Computer Science, 2003. 2, 6
7. A. E. Ibrahim. *Genshade: an Evolutionary Approach to Automatic and Interactive Procedural Texture Generation.* Doctoral Thesis, College of Architecture, A&M University, 1998. 2
8. J. Kautz and M. McCool. Interactive Rendering with Arbitrary BRDFs Using Separable Approximations. *Rendering Techniques '99 (Proc. of Eurographics Workshop on Rendering)*, pp. 281–292, 1999. 3
9. J. Kautz, P.-P. Sloan, and J. Snyder. Fast, Arbitrary BRDF Shading for Low-Frequency Lighting Using Spherical Harmonics. *Proc. of the 12th Eurographics Workshop on Rendering*, pp. 301–308, 2002. 3
10. J. R. Koza. Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. *Proc. of the 11th Int. Conf. on Genetic Algorithms (San Mateo)*, pp. 768–774, 1989. 2
11. F. Kühling, K. Wolff, and P. Nordin. Brute-Force Approach to Automatic Induction of Machine Code on CISC Architectures. *Genetic Programming, Proc. of the 5th European Conf. (Kinsale)*, pp. 288–297, 2002. 2
12. L. Latta and A. Kolb. Homomorphic Factorization of BRDF-based Lighting Computation. *ACM Transactions on Graphics (Proc. SIGGRAPH 2002)*, **21**(3):509–516, 2002 3
13. X. Liu, Y. Yu, and H.-Y. Shum. Synthesizing Bidirectional Texture Functions for Real-World Surfaces. *ACM Computer Graphics (Proc. of SIGGRAPH 2001)*, pp. 97–106, 2001. 6
14. J. Loviscach and J. Meyer-Spradow. Genetic Programming of Vertex Shaders. *Proc. of EuroMedia 2003 (Plymouth)*, pp. 29–31, 2003. 2, 5, 6
15. T. Malzbender, D. Gelb, and H. Wolters. Polynomial Texture Maps. *ACM Computer Graphics (Proc. of SIGGRAPH 2001)*, pp. 519–528, 2001. 3
16. M. D. McCool, J. Ang, and A. Ahmad. Homomorphic Factorization of BRDFs for High-Performance Rendering. *ACM Computer Graphics (Proc. of SIGGRAPH 2001)*, pp. 171–178, 2001 3
17. J. F. Miller and P. Thomson. Cartesian Genetic Programming. *Genetic Programming, Proc. of EuroGP 2000 (Edinburgh)*, pp. 121–132, 2000. 2
18. B.-T. Phong. Illumination for Computer-Generated Pictures. *Communications of the ACM*, **1**8(6):311–317, 1975. 2
19. R. Ramamoorthi and P. Hanrahan. Frequency Space Environment Map Rendering. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2002)*, **21**(3):517–525, 2002. 3
20. K. Sims. Artificial Evolution for Computer Graphics. *Computer Graphics*, **25**(4):319–328, 1991. 2
21. T. Soule and R. B. Heckendorn. An Analysis of the Causes of Code Growth in Genetic Programming. *Genetic Programming and Evolvable Machines*, **3**:283–309, 2002. 5
22. A. L. Wiens and B. J. Ross. Gentropy: Evolving 2D Textures. *Computers & Graphics*, **26**:75–88, 2002. 2