Sampling Procedural Shaders Using Affine Arithmetic

WOLFGANG HEIDRICH, PHILIPP SLUSALLEK, and HANS-PETER SEIDEL
University of Erlangen, Computer Graphics Group

Procedural shaders have become popular tools for describing surface reflectance functions and other material properties. In comparison to fixed resolution textures, they have the advantage of being resolution-independent and storage-efficient.

While procedural shaders provide an interface for evaluating the shader at a single point, it is not possible to easily obtain an average value of the shader together with accurate error bounds over a finite area. Yet the ability to compute such error bounds is crucial for several interesting applications, most notably hierarchical area sampling for global illumination, using the finite element approach, and for generation of textures used in interactive computer graphics.

Using affine arithmetic for evaluating the shader over a finite area yields a tight, conservative error interval for the shader function. Compilers can automatically generate code for utilizing affine arithmetic from within shaders implemented in a dedicated language such as the RenderMan shading language.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*error analysis*, *interval arithmetic*; G.1.4 [**Numerical Analysis**]: automatic differentiation; I.3.7 [**Computer Graphics**]: Quadrature and Numerical Differentiation—*color*, *shading*, *shadowing and texture-radiosity*; I.4.1 [**Image Processing and Computer Vision**]: Digitization—*sampling*; I.4.7 [**Image Processing and Computer Vision**]: Feature Measurement—*texture*

General Terms: Experimentation, Graphics, Performance, Theory, Verification

Additional Key Words and Phrases: Affine arithmetic

1. INTRODUCTION

The ability to compute mean reflectance coefficients as well as error bounds for a shader over a finite area of a surface has several interesting applications. For example, in radiosity computations [Cohen and Wallace 1993], the mean reflectance of surface patches is required for setting up a linear equation system for the global illumination problem. Hierarchical radiosity [Hanrahan and Salzman 1989] adaptively subdivides the patches in order to compute interactions between patches exchanging large quantities of radiosity or energy [Lischinski et al. 1994] with higher precision. The

known subdivision criteria assume a constant reflection coefficient for each patch. The ability to compute conservative error bounds for the reflection function over a patch would allow for improved subdivision criteria based not only on the amount of energy, but also on the amount of detail in the reflection function.

Another application for area samples with conservative error bounds is the generation of texture maps in cases where procedural shaders cannot be directly supported by the renderer, either due to a limitation of the renderer (for example, renderers based on libraries like OpenGL), or due to performance penalties. With the help of error bounds on reflectance values it is possible to generate hierarchically a precomputed texture from procedural shaders. Starting with a coarsely sampled texture area, we recursively refine those area samples in which the error is above a given threshold.

This adaptive subdivision yields a piecewise constant approximation of the shader, where the error in each cell is smaller than the threshold. This is the case if either the sampling rate is higher than the highest frequency of the shader (that is, there are no frequencies above the Nyquist limit), or the amplitudes of these higher frequencies are small enough to be neglected. Using this kind of hierarchical analysis, the number of samples can be minimized without losing any detail.

Since procedural shaders offer only a point sampling interface, the only way to generate error estimates for a shader function over a finite area is Monte Carlo sampling. However, this method only yields an estimate of the true error bounds, and since Monte Carlo methods only converge with $O(\sqrt{N})$ [Kalos and Whitlock 1986], this process can be very expensive. For applications where truly conservative bounds are required, these methods cannot be used at all.

In the past, Greene and Kass [1994] have used interval arithmetic for anti-aliasing shaders. These shaders are programmed in a visual dataflow language [Kass 1992], which is then compiled to C++.

In this paper, we describe a general method for computing tight, conservative error bounds for procedural RenderMan shaders using affine arithmetic. Using this method, it is possible to apply procedural shaders to the application domains mentioned above. We first give a brief overview of affine arithmetic in general before we describe the details of applying it to procedural shaders.

2. AFFINE ARITHMETIC

Affine arithmetic (AA), first introduced in Comba and Stolfi [1993], is an extension of interval arithmetic [Moore 1966]. It has been successfully applied to several problems for which interval arithmetic had been used before [Musgrave et al. 1989; Snyder 1992a; 1992b]. This includes reliable intersection tests of rays with implicit surfaces, and recursive enumerations of implicit surfaces in quad-tree-like structures [Figueiredo 1996; Figueiredo and Stolfi 1996].

Like interval arithmetic, AA can be used to manipulate imprecise values and to evaluate functions over intervals. It is also possible to keep track of truncation and round-off errors. In contrast to interval arithmetic, AA also maintains dependencies between the sources of error, and thus manages to compute significantly tighter error bounds. Detailed comparisons between interval arithmetic and affine arithmetic can be found in Comba and Stolfi [1993]; Figueiredo [1996]; and Figueiredo and Stolfi [1996].

Affine arithmetic operates on quantities known as *affine forms*, given as polynomials of degree one in a set of *noise symbols* ϵ_i.

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \cdots + x_n\epsilon_n.$$

The coefficients x_i are known real values, while the values of the noise symbols are unknown but limited to the intervals $\mathbf{U} := [-1, 1]$. Thus, if all noise symbols can independently vary between -1 and 1, the range of possible values of an affine form $\hat{x}$ is

$$[\hat{x}] = [x_0 - \xi, x_0 + \xi], \quad \xi = \sum_{i=1}^{n} |x_i|.$$

Computing with affine forms consists of replacing each elementary operation $f(x)$ on real numbers with an analogous operation $f^*(\epsilon_1, \ldots, \epsilon_n) := f(\hat{x})$ on affine forms.

If f is itself an affine function of its arguments, we can apply normal polynomial arithmetic to find the corresponding operation f^*. For example, we get

$$\hat{x} + \hat{y} = (x_0 + y_0) + (x_1 + y_1)\epsilon_1 + \cdots + (x_n + y_n)\epsilon_n$$

$$\hat{x} + \alpha = (x_0 + \alpha) + x_1\epsilon_1 + \cdots + x_n\epsilon_n$$

$$\alpha\hat{x} = \alpha x_0 + \alpha x_1\epsilon_1 + \cdots + \alpha x_n\epsilon_n$$

for affine forms $\hat{x}$, $\hat{y}$, and real values α.

2.1 Nonaffine Operations

If f is not an affine operation, the corresponding function $f^*(\epsilon_1, \ldots, \epsilon_n)$ cannot be exactly represented as a linear polynomial in the ϵ_i. In this case, it is necessary to find an affine function $f^a(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n$ approximating $f^*(\epsilon_1, \ldots, \epsilon_n)$ as well as possible over $\mathbf{U}^n$. An additional *new* noise symbol ϵ_k has to be added to represent the error introduced by this approximation. This yields the following affine form for the operation $z = f(x)$:

$$\hat{z} = f^a(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n + z_k\epsilon_k, k \notin \{1, \ldots, n\}.$$

The coefficient z_k of the new noise symbol has to be an upper bound for the error introduced by the approximation of f^* with f^a:

$$z_k \geq \max\{|f^*(\epsilon_1, \ldots, \epsilon_n) - f^a(\epsilon_1, \ldots, \epsilon_n)| : \epsilon_i \in \mathbf{U}\}.$$

For example, it turns out (see Comba and Stolfi [1993] for details) that a good approximation for the multiplication of two affine forms $\hat{x}$ and $\hat{y}$ is

$$\hat{z} = x_0 y_0 + (x_0 y_1 + y_0 x_1)\epsilon_1 + \cdots + (x_0 y_n + y_0 x_n)\epsilon_n + uv\epsilon_k,$$

with $u = \Sigma_{i=1}^{n} |x_i|$ and $v = \Sigma_{i=1}^{n} |y_i|$. In general, the best approximation f^a of f^* minimizes the Chebyshev error between the two functions.

The generation of affine approximations for most of the functions in the standard math library is relatively straightforward. For a univariate function $f(x)$, the isosurfaces of $f^*(\epsilon_1, \ldots, \epsilon_n) = f(x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n)$ are hyperplanes of $\mathbf{U}^n$ that are perpendicular to the vector $(x_1, \ldots, x_n)$. Since the isosurfaces of every affine function $f^a(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n$ are also hyperplanes of this space, it is clear that the isosurfaces of the best affine approximation f^a of f^* are also perpendicular to $(x_1, \ldots, x_n)$. Thus, we have

$$f^a(\epsilon_1, \ldots, \epsilon_n) = \alpha\hat{x} + \beta = \alpha(x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n) + \beta$$

for some constants α and β. As a consequence, the minimum of $\max_{\epsilon_i \in \mathbf{U}} |f^a - f^*|$ is obtained by minimizing $\max_{\epsilon_i \in \mathbf{U}} |f(\hat{x}) - \alpha\hat{x} - \beta| = \max_{x \in [a,b]} |f(x) - \alpha x - \beta|$, where $[a, b]$ is the interval $[\hat{x}]$. Thus, approximating f^* has been reduced to finding a linear Chebyshev approximation for a univariate function, which is a well-understood problem [Cheney 1966]. An example for such an approximation is outlined in the Appendix. A compilation of affine approximations for common math library functions can be found in Heidrich [1997].

Most multivariate functions can be handled by reducing them to a composition of univariate functions. For example, the maximum of two numbers can be rewritten as $\max(x, y) = \max_0(x - y) + y$, with $\max_0(z) := \max(z, 0)$. For the univariate function $\max_0(z)$, we can use the scheme above.

3. APPLICATION TO PROCEDURAL SHADERS

In order to apply AA to procedural shaders, it is necessary to investigate which additional features are provided by shading languages in comparison to standard math libraries. In the following, we restrict ourselves to the functionality of the RenderMan shading language [Harahan and Lawson 1990; Pixar 1989; Upstill 1990], which is generally agreed to be one of the most flexible languages for procedural shaders. Since its features are a superset of most other shading languages (for example, Alias/Wavefront [1996] and Molnar et al. [1992]), the concepts apply to these other languages as well.

Shading languages usually introduce a set of specific data types and functions exceeding the functionality of general-purpose languages and libraries. Most of these additional functions can be easily approximated by affine forms using techniques similar to the ones outlined in the previous section. Examples of this kind of domain-specific function are continuous and discontinuous transitions between two values, like step functions, clamping of a value to an interval, or smooth Hermite interpolation between two values.

The more complicated features include splines, pseudo-random noise, and derivatives of expressions. The latter two are discussed in Sections 3.2 and 3.3 in detail.

New data types in the RenderMan shading language are points and color values; both are simply vectors of scalar values. Affine approximations of the typical operations on these data types (sum, difference, scalar-, dot-, and cross product, as well as the vector norm) can easily be implemented based on the primitive operations on affine forms.

Every shader in the RenderMan shading language is supplied with a set of explicit, shader specific parameters that may be linearly interpolated over the surface, as well as fixed set of implicit parameters (global variables). The implicit parameters include the location of the sample point, the normal and tangents in this point, as well as vectors pointing towards the eye and the light sources. For parametric surfaces, these values are functions of the surface parameters u and v, as well as the size of the sample region in the parametric domain du and dv.

For parametric surfaces, including all geometric primitives defined by the RenderMan standard, the explicit and implicit shader parameters can therefore be computed by evaluating the corresponding function over the affine forms for u, v, du, and dv. The affine forms of these four values have to be computed from the sample region in parameter space. For many applications, du and dv are actually real values on which the affine forms of u and v depend: $\hat{u} = u_0 + du \cdot \epsilon_1$ and $\hat{v} = v_0 + dv \cdot \epsilon_2$.

3.1 Control Statements

Like most other programming languages, shading languages provide a set of control statements. These can be grouped into two categories: conditional statements and loops. Both use Boolean expressions composed of equalities and inequalities to choose between different execution paths.

In the context of affine arithmetic or interval arithmetic, the use of inequalities has the problem that as soon as two values with overlapping ranges are compared, the inequality is neither true nor false. Therefore, some form of three-state logic or fuzzy logic has to be used. In case an expression is not decidable, both execution paths have to be executed and the results have to be merged.

We have developed a method to deal with this problem by replacing each inequality with a step function. For example, $x < y$ becomes $\mathbf{step}(y - x)$, where $\mathbf{step}(x) := 0$ for $x < 0$ and $\mathbf{step}(x) := 1$ otherwise. It is

straightforward to find an optimal affine approximation for this step function. Because of the discontinuity, there is a whole family of linear functions passing through the point (0, 1/2), for which the Chebyshev error is 1/2, the optimum value. One of these functions is $f(\hat{x}) = 1/2$, so that $\hat{z} := 1/2 + 1/2\epsilon_k$ is an optimal implementation of the **step**( ) function.

After this replacement, Boolean expressions in inequalities can be written as arithmetic expressions. For example, $x_1 < y_1 \; and \; x_2 < y_2$ becomes **step**$(y_1 - x_1) \cdot$ **step**$(y_2 - x_2)$. The following example is an excerpt of the "screen" shader [Upstill 1990] using an **if**-statement to compute the opacity value.

```
if(mod(s, 1.0) < density || mod(t, 1.0) < density )
  Opacity:= 1.0;
else
  Opacity:= 0.0;
endif
```

Using the transformations for Boolean expressions as described above, the **if**-statement is translated into the following piece of code. Loops are treated in a similar fashion.

```
condition:= 1.0 − (1.0-step(density-mod(s,1.0))) * (1.0-step(density-mod(t,1.0)));
(min,max)= range(condition);
if(min>= 1.0)
  Opacity:= 1.0;
else if(max<= 0.0)
  Opacity:= 0.0;
else
  Opacity1:= 1.0; Opacity2:= 0.0;
  Opacity:= condition*Opacity1 + (1-condition)*Opacity2;
endif
```

The variables Opacity1 and Opacity2 hold the temporary values of the opacity for the two execution paths. This kind of temporary variable has to be introduced for all variables that may change in either the **if** or the **else** part of the conditional.

Figure 1 shows an image of the screen shader including the illumination effects from a single point light source, sampled on a 512×512 grid using this substitution. Each pixel represents an area sample of the corresponding grid cell. The left side of the figure shows the color-coded range size for the affine form of each color value. Darker color values mean larger range sizes. Ninety-one percent of the color values have errors of less than 5 percent. Still, 90 percent have an error smaller than 1 percent.

It is clear that high errors occur only at locations with high color gradients, where the sample area covers both bright and dark areas of the shader. The oval region in the right half of the texture image represents the front-facing part of the sphere. For points outside this region, the normal vectors are inverted to point towards the eye. This, together with the high specular component of the shader, results in a larger color gradient.

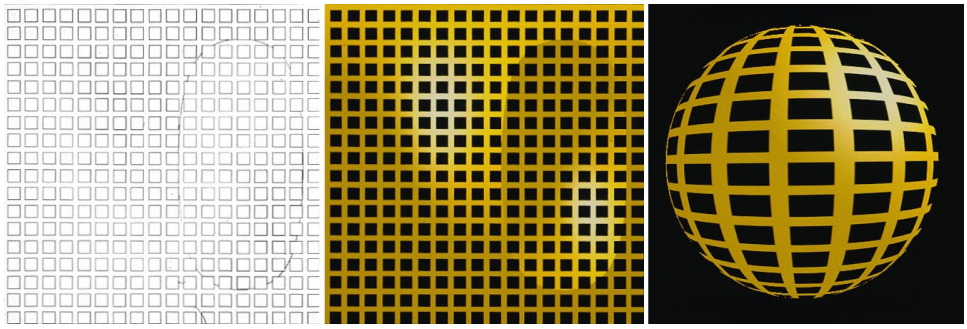

Fig. 1. The screen shader, including illumination effects, sampled on a sphere. The **if**-statements in the shader have been replaced by step functions, as described in the text. The left image shows the size of the error interval as a color-coded error plot (dark regions mean large errors), the center image shows the sampled texture, and the right image shows the texture mapped onto a sphere.

3.2 Noise and Turbulence

In order to generate interesting-looking detail, procedural shaders require a pseudo-random function which is both reproducible and continuous. This function is typically called **noise** [Ebert et al. 1994]. Many algorithms for noise functions have been proposed, starting with value noise, which interpolates pseudo-random data values at the grid points of an integer lattice [Lewis 1989], over gradient noise [Perlin 1985; Perlin and Hoffert 1989], which enforces a pseudo-random gradient at the lattice points, to hybrid methods [Ward 1991] and sparse convolution noise [Lewis 1989].

The choice of a specific noise function has a significant impact on the characteristics of the shader. In particular, the base frequency of the **noise** influences the size of the features and irregularities on the shader. For proper shader-driven anti-aliasing, this base frequency has to be known to the shader [Ebert et al. 1994]. Because most published RenderMan shaders implicitly assume gradient noise as introduced by Perlin and Hoffert [1989], we decided to use this version of the **noise** function, too.

To evaluate gradient noise at a sample point in 3-dimensional space, first pseudo-random gradient vectors and scalar values are generated in each vertex of an integer lattice. At each vertex (x_0, y_0, z_0), the pair of gradient g and scalar d defines a linear function f with $f(x_0, y_0, z_0) = d$ and $\nabla f(x_0, y_0, z_0) = g$. The eight linear functions corresponding to the vertices next to the sample point are combined using smoothed trilinear interpolation [Ebert et al. 1994]:

```
gradientNoise(x, y, z)
begin
  ix := floor(x);iy := floor(y);iz := floor(z);
  fx := x − ix; fy := y − iy; fz := z − iz;
  wx := smoothstep(fx);wy := smoothstep(fy);wz := smoothstep(fz);
  look up gradients g000, g001, . . . 111 and d000, d001, . . . , d111 in points
    (ix, iy, iz), (ix, iy, iz + 1), . . . , (ix + 1, iy + 1, iz + 1).
```

each pair of gradient and scalar defines a linear function:
$f_{ijk}(fx, fy, fz) := \langle g_{ijk} | (fx - i, fy - j, fz - k) \rangle + d_{ijk}, \quad i, j, k \in \{0, 1\}$
trilinear interpolation of the $f_{ijk}(fx, fy, fz)$, using wx, wy, and wz as weights.
end

In this algorithm, **smoothstep**$(x) := 3x^2 - 2x^3$ defines a smooth transition between 0 and 1 over the interval [0, 1]. An affine approximation for the **noise** function over a single lattice cell can be derived quite easily by replacing the real valued versions of **smoothstep**, the linear functions, and the trilinear interpolation by functions operating on affine forms.

It is, however, significantly harder to find a good affine approximation if the vector of affine forms representing the 3-dimensional point of evaluation spans multiple lattice cells. In this case we decided to fall back to interval arithmetic for the evaluation of the noise function. The resulting interval is stored as an affine form $z_0 + z_k \epsilon_k$, so that subsequent computations can still use AA to keep track of the dependency of other expressions on the error introduced by the **noise** function.

If the range of one of the components of the point of evaluation spans more than two lattice cells, we simply return the maximum interval of the noise function ([0, 1] in the RenderMan shading language) as an affine form. Although this approach might seem overly simplified at first glance, it does not usually have a strong impact on the quality of the error bounds, since in this case the sample contains frequencies above the Nyquist limit. Moreover, a few grid cells are usually enough for the noise function to use its full dynamic range. On the other hand, many of the more involved shaders avoid this problem altogether by removing frequencies above the sampling rate using clamping [Norton et al. 1982]. Also, in algorithms that hierarchically subdivide the parameter range, the ranges of all components eventually span no more than two adjacent grid cells.

More care has to be taken in cases where the ranges of the components only span up to two adjacent cells in either direction. In a hierarchical algorithm, this case occurs no matter how finely the parameter range has been subdivided. If in this case the full interval [0, 1] is returned, the shader bounds will not converge around points located on the boundaries of noise grid cells.

Therefore, we restrict the range of the point to each of the covered cells and evaluate **noise** over the restricted intervals. The ranges of the resulting affine forms are combined for the final result. A potential problem with this approach is that, although the range for the resulting affine form is a relatively tight bound for the true range of the noise function around the sample point, the dependencies of the value on the noise symbols are lost. This can lead to problems when computing derivatives, as described in Section 3.3. In shaders without derivatives, we found that this method produces very tight error bounds; see Figure 2, where the "wood" shader [Upstill 1990] is applied to a sphere.

Fig. 2. The wood shader uses the **noise** function to generate irregularities in the rings.

The example shows again that errors occur only in areas of high gradients in the shader function and on the transition from lit to unlit areas. This is also confirmed by the statistics: 98 percent of the pixel values have an error below 5 percent, and 93 percent of the errors are below 1 percent.

While the wood shader uses **noise** at a fixed frequency only, a lot of other shaders use it to generate a stochastic spectral function with a specific frequency/power spectrum. Such a function is given as Σ_m **noise**$(P \cdot f^m)/f^m$ for some scaling factor f. This function is called *fractal noise*, or *turbulence*. One of the shaders using turbulence is the "blue marble" shader [Upstill 1990], which is shown at the top of Figure 3.

The blue marble shader uses clamping to eliminate high frequencies, but still contains frequencies that are close to the sampling rate. This leads to relatively large gradients, and thus to larger error intervals. In the example shown in Figure 3, 81 percent of the pixels lie within 5 percent, and 59 percent within 1 percent of error. These numbers reflect the fact that the true variation of the shading function over each pixel is relatively high, which can be seen from the cross section of the blue marble shader in the lower left of Figure 3. The error bounds produced by AA are still relatively tight around the true variation of the shader over each sample area.

A single supersampling step for this function, that is, an increase of the sampling rate by a factor of two without introducing even higher frequencies into the shader function, yields substantially improved error bounds (right side of Figure 3). The statistics for this case are 92 percent of the values below 5 percent and 82 percent below 1 percent error.

3.3 Derivatives

Another feature unique to shading languages is the possibility of calculating the derivatives of expressions. The RenderMan shading language is particularly flexible in this respect, since it does not limit the kind of expressions that can be derived, nor the location in the shader code where a derivative can be computed. This can often cause severe problems in implementations [Slusallek et al. 1994].

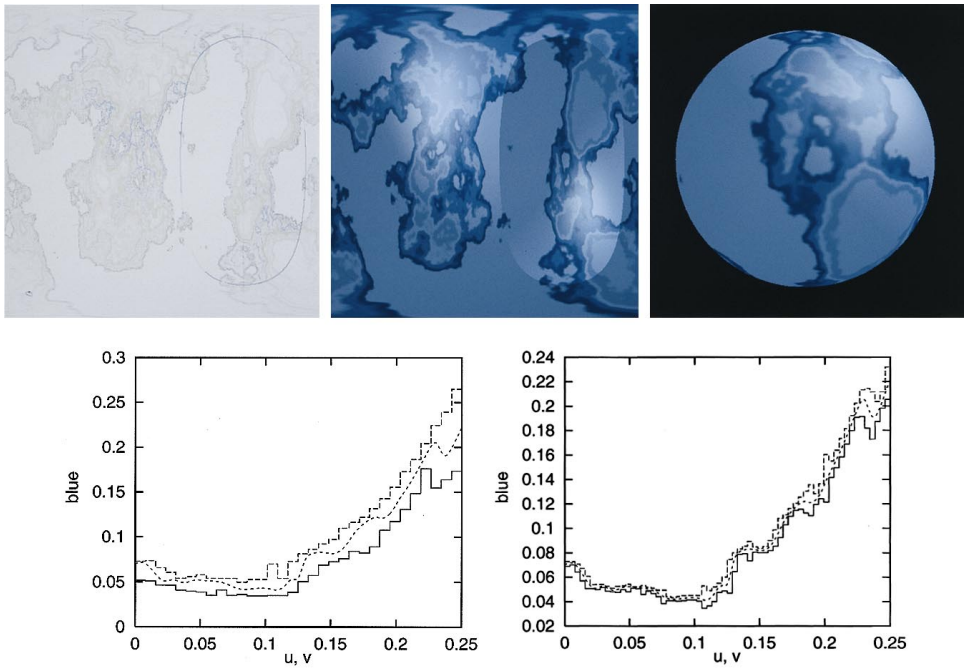

Fig. 3. Top: The blue marble shader. Bottom: Cross sections of the shader sampled at the original sampling rate (left) and twice this sampling rate (right) over the interval $u \epsilon [0, 0.25]$ with $v = 0.3$. The cross sections show the blue channel of the shader.

Among the most common reasons for using derivatives is the calculation of tangent vectors and normal vectors at a particular point:

$$\mathbf{calculatenormal}(P) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}.$$

Note that the point P does not need to be a point on the original surface, but can be a displaced point generated by a bump-mapping algorithm. A related application of derivatives is the calculation of the area of the differential surface element around a point, which is required for proper shader-driven anti-aliasing.

$$\mathbf{area}(P) = \|\mathbf{calculatenormal}(P)\|.$$

In the RenderMan shading language, derivatives are supported in the form of divided differences. For example, derivatives of arbitrary expressions with respect to the change in surface parameters u and v are defined as

$$\mathbf{Du}(f(u)) := \frac{f(u + du) - f(u)}{du} \quad \text{and} \quad \mathbf{Dv}(f(v)) := \frac{f(v + dv) - f(v)}{dv}.$$

Derivatives with respect to arbitrary expressions are computed using the chain rule: $\mathbf{Deriv}(f, g) := \mathbf{Du}(f)/\mathbf{Du}(g) + \mathbf{Dv}(f)/\mathbf{Dv}(g)$.

Fig. 4. A sphere with a simple bump-mapped surface. The normal vector has been computed using derivatives with respect to the parametric directions u and v.

With these definitions, conservative bounds for derivatives according to the RenderMan standard can be obtained by maintaining the triple $(\hat{x}(u, v), \hat{x}^u := \hat{x}(u + du, v), \hat{x}^v := \hat{x}(u, v + dv))$ for each expression $\hat{x}(u, v)$ during the execution of the shader. The divided difference of each expression is then available at every time. Each expression of the shader has to be evaluated at all three points (u, v), $(u + du, v)$, and $(u, v + dv)$; and of course the parameters of the shader have to be provided at all three points as well.

Note that for this algorithm to work with **if**-statements, both the **if**- and the **else** path have to be executed if the Boolean control expressions for the three points yield different results. A similar problem occurs with loops where the number of iterations may depend on all three different expressions. If the derivatives are not used within the loop itself, we can simply generate three independent loops for each of the three parametric points. If derivatives are used within loops, our current implementation ignores the problem, and terminates the recursion according to the value of the main expression. On these rare occasions, this can result in situations where the bounds on the derivatives are not conservative.

An example of a simple bump-mapped shader implemented using this approach can be seen in Figure 4. The image shows a sphere modulated with a sine wave in one parametric direction. The new shading normal has been computed using the function **calculatenormal**, with the displaced surface point, as described above. More than 98 percent of the pixel values are within 5 percent of error, and 94 percent of the pixels have an error smaller than 1 percent. The error plot shows that the largest errors are due to singularities at the poles of the sphere. Due to bump mapping, there are also several areas in this shader for which the normal is inverted with respect to the eye. Around these areas there are also fine lines of medium-sized errors.

The method of divided differences works fairly well for relatively simple expressions, but the errors become larger as the complexity of the expressions grows. The quality of the bounds degrades distinctly when **noise** is used at high frequencies as, for example, with the “eroded” shader [Upstill 1990], shown in Figure 5.

Fig. 5. A sphere with "encoded" shader applied.

It can be seen from the error plot in Figure 5 that large errors occur at the boundaries of **noise** lattice cells, due to the way **noise** values for affine forms that span multiple grid cells are computed. Since divided differences use differences of highly correlated values, it is important for the affine forms to reflect this correlation. However, with the current method of computing **noise** this correlation is lost at grid boundaries. Roughly 77 percent of the pixel values are within 5 percent of error, and 53 percent of the pixels have an error smaller than 1 percent. These values could certainly be improved with a better affine approximation of the **noise** function.

Alternative Ways to Compute Derivatives

Conformance with the RenderMan standard is a major reason for using divided differences as an approximation for derivatives. If this is not an important issue, we can use other ways to implement derivatives. One is to use information contained in the affine forms as an estimate for derivatives directly. In Section 3 we mentioned that du and dv are often real values, and u and v depend on them: $\hat{u} = u_0 + du \cdot \epsilon_1$ and $\hat{v} = v_0 + dv \cdot \epsilon_2$. In this case we can use the coefficients z_1 and z_2 of any affine form $\hat{z} = f(\hat{u}, \hat{v})$ as an estimate for its partial derivatives $\partial f/\partial u$ and $\partial f/\partial v$. If f is differentiable, the mean value of $\partial f/\partial u$ is contained in the interval $z_1 \pm 1/|du| \Sigma_3^n z_i$. This does not yield conservative bounds for the derivative over the whole range of u and v, but neither do divided differences. This approach is obviously much faster than divided differences, where each expression has to be evaluated three times.

However, this way of estimating derivatives causes problems in discontinuous areas, which occur quite frequently in procedural shaders. As an example, consider the **step**( ) function mentioned above. If the range of $\hat{u}$ contains zero, the true range of derivatives for ∂**step**$(u)/\partial u$ is $[0, \infty)$. Divided differences yield the range $[0, 1/du]$, which converges to the correct range as the parameter domain is subdivided. The approach described above, however, yields the range $[-1/du, 1/du]$, which is not useful in practice. Similar problems can even arise in continuous areas of the shader, for example at grid boundaries of the **noise** function. Since we

use interval arithmetic to evaluate the **noise** function in these areas, the resulting affine forms do not depend on ϵ_1 and ϵ_2 at all: **noise**$(\hat{x}) = z_0 + z_k\epsilon_k$. Thus the resulting range for the derivative is $[-|z_k|/du, |z_k|/du]$, which does not, in general, converge to a single scalar value.

Another possible algorithm for implementing derivatives is *automatic differentiation* [Rall 1981]. In contrast to the other two methods, automatic differentiation computes conservative bounds for the derivatives in the mathematical sense. Instead of maintaining $\hat{x}^u$ and $\hat{x}^v$, automatic differentiation directly stores the partial derivatives $\partial\hat{x}/\partial u$ and $\partial\hat{x}/\partial v$ of every expression. These partial derivatives are computed by applying the chain rule to every primitive expression. For example, the logarithm of the triple $(\hat{x}, \partial\hat{x}/\partial u, \partial\hat{x}/\partial v)$ is $(ln(\hat{x}), (\partial\hat{x}/\partial u)/\hat{x}, (\partial\hat{x}/\partial v)/\hat{x})$. This algorithm also avoids the problems we have with the combination of derivatives and **for**-loops when using divided differences.

For normal floating point arithmetic, automatic differentiation is numerically more stable than divided differences. It is not clear whether this also applies to AA, since the expressions for derivatives tend to be more complex, and thus a larger error could be introduced by the larger number of nonaffine operations. However, this is certainly a point worth investigating in the future.

Of course, both divided differences and automatic differentiation introduce a significant amount of computational overhead, and thus the shading language compiler should optimize the code to compute $\hat{x}^u$ and $\hat{x}^v$ or $\partial\hat{x}/\partial u$ and $\partial\hat{x}/\partial v$ only for expressions eventually used in a derivative. This requires the compiler to perform some sort of dependency analysis, as discussed by Guenter et al. [1995].

4. RESULTS

In this paper we use affine arithmetic to obtain conservative bounds for shader values over a parameter range. In principle, we could also use any other range analysis method for this purpose. It is, however, important that the method generates tight, conservative bounds for the shader. Conservative bounds are important so that no small detail is missed, while tight bounds reduce the number of subdivisions, and therefore save both computational time and memory.

We have performed tests to compare interval arithmetic to affine arithmetic for the specific application of procedural shaders. Our results show that the bounds produced by interval arithmetic are significantly wider than the bounds produced by affine arithmetic. Figure 6 shows the wood shader sampled at a resolution of 512×512. The error plots show that interval arithmetic yields errors up to 50 percent in areas where affine arithmetic produces errors below 1/256. As a consequence the textures generated from this data, by assigning the mean values of the computed range to each pixel, reveal severe artifacts in the case of interval arithmetic.

Fig. 6. The wood shader sampled at a resolution of 512 × 512. From left to right: Error plot using interval arithmetic, resulting texture, error plot using affine arithmetic, resulting texture.

The corresponding error histogram in Figure 7 shows that while most of the per-pixel errors for affine arithmetic are less than 3 percent, most of the errors for interval arithmetic are in the range of 5–10 percent, and a significant number is even higher (up to 50 percent).

These results are not surprising. All the expressions computed by a procedural shader depend on four input parameters: u, v, du, and dv. Affine arithmetic keeps track of most of these subtle dependencies, while interval arithmetic ignores them completely. The more complicated functions become, the more dependencies between the sources of error exist, and the bigger the advantage of AA. These results are consistent with prior studies in Comba et al. [1993], Figueiredo [1996], and Figueiredo and Stolfi [1996].

Both affine and interval arithmetic bounds can be further improved by finding optimal approximations for larger blocks of code, instead of just library functions. This process, however, requires human intervention and cannot be done automatically.

The method presented in this paper is thus the only practical choice, as long as conservative error bounds are required. Other applications, for which an estimate of the bounds is sufficient, could also use Monte Carlo sampling. In this case it is interesting to analyze the number of Monte Carlo samples and the resulting quality of the estimate that can be obtained in the same time as a single area sample using AA. Table I compares these numbers in terms of floating point operations (FLOPS) and execution time (on a 100MHz R4000 Indigo) for the various shaders in this paper.

The relative performance of AA decreases for more complicated shaders, since more error variables are introduced due to the increased amount of nonaffine operations. Table I shows that depending on the shader, 5 to 10 point samples are as expensive as a single AA area sample. To see what this means for the quality of the bounds, consider the screen shader with a density of 0.5. This density means that 75 percent of the shader is opaque, while 25 percent is translucent. If we take 7 point samples of this shader, which is about as expensive as a single AA sample, the probability that all samples compute the same opacity is $0.75^7 + 0.25^7 \approx 13.4$ percent. Even with 10 samples the probability is still 5.6 percent.

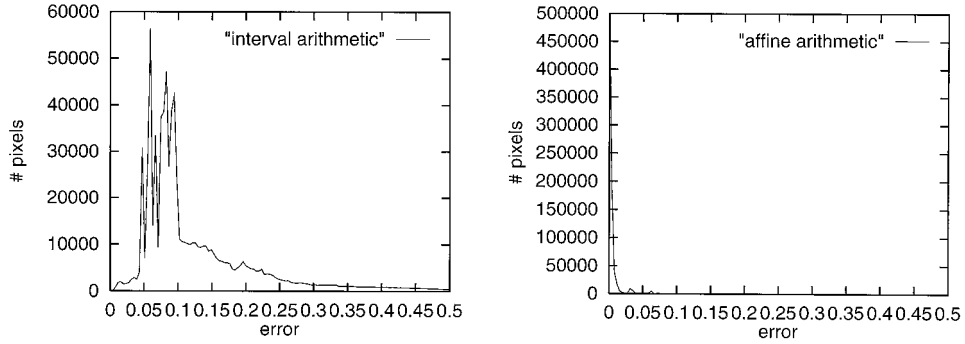

Fig. 7. Error histograms for the wood shader for interval arithmetic (left) and affine arithmetic (right).

Table I. FLOPS per Sample and Timings for 4096 Samples (for stochastic point sampling (ps) and AA area sampling (aa).

Shader	FLOPS (ps)	FLOPS (aa)	Ratio	Time (ps)	Time (aa)	Ratio
screen	24	214	1:8.92	4.57	33.48	1:7.32
wood	803	6738	1:8.39	8.34	86.53	1:10.38
marble	4386	28812	1:6.57	9.46	88.52	1:9.36
bumpmap	59	487	1:8.25	3.76	20.43	1:5.43
eroded	2995	26984	1:9.01	18.85	193.33	1:10.27

For the example that uses area samples as a subdivision criterion in hierarchical radiosity, this means that a wall covered with the screen shader has a probability of 13.4 (or 5.6) percent of not being subdivided at all. The same probability applies to each level in the subdivision hierarchy independently. These numbers indicate that AA is superior to point sampling, even when only coarse estimates of the error bounds are desired.

5. APPLICATIONS AND CONCLUSION

We have presented a method to generate conservative error bounds for area samples of procedural shaders. The algorithm is based on affine arithmetic and works by evaluating the compiled shader using affine forms instead of normal floating point values. We have described how the functionality of the popular RenderMan shading language can be implemented with affine arithmetic and how affine approximations of the various functions can be generated.

We have implemented a hierarchical subdivision scheme for procedural RenderMan shaders using the methods described in this paper. Given a tolerance value, the algorithm hierarchically subdivides the parameter domain of the shader, until the area samples for all subdivision cells are within the tolerance, or a maximum recursion level is reached. This algorithm results in a hierarchical analysis of the shader and allows us to find an optimal resolution for representing the resulting texture. Figure 8

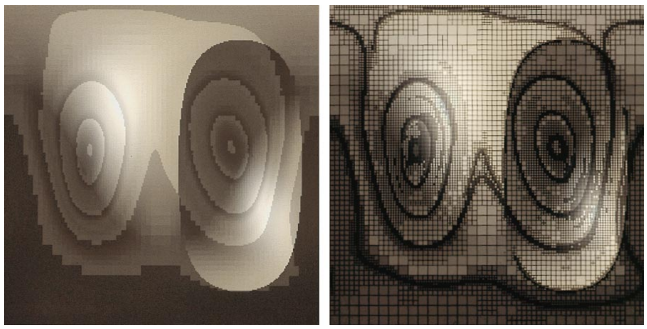

Fig. 8. The hierarchically subdivided wood shader for a maximum resolution of 256 × 256. Cells with an error of over 5 percent were subdivided at each level. This resulted in a total of 19512 samples instead of 65536 for the full resolution (29.8 percent).

shows the result of such a subdivision for the wood shader with a tolerance of 5 percent and a maximum recursion depth of 8 (resolution 256 × 256).

Using this hierarchical subdivision scheme, it is possible to precompute textures from procedural shaders with an unknown amount of detail information. These textures can then be applied (for example, with the OpenGL renderer we used to generate the images in this paper). We are currently working on integrating the subdivision scheme into the Vision rendering system [Slusallek and Seidel 1995] for supporting finite element-based global illumination. The level of subdivision shown in Figure 8 is sufficient for this purpose, but clearly the tolerance has to be decreased for textures directly used in OpenGL renderers.

In Stamminger et al. [1997], an algorithm is described for bounding form factors and lighting computations. This yields conservative bounds for both the geometric and the lighting computations in a radiosity system, but ignores potential variations of surface materials. In this sense, our method for bounding the error for area samples of procedural shaders now fills in the missing parts for bounds on global illumination simulations. The two algorithms combined mean that neither geometric nor material details can be missed. We think that the ability to generate conservative bounds for procedural shaders will also be useful in a variety of other applications, such as ray-tracing of displacement shaders.

APPENDIX. An Affine Approximation of $f(x) = 1/x$

In Section 2.1 we saw that the best affine approximation of a function $f^*(\epsilon_1, \ldots, \epsilon_n) = f(\hat{x})$ is $f^a(\epsilon_1, \ldots, \epsilon_n) = \alpha\hat{x} + \beta$, for some α and β.

As an example, consider the function $f(x) = 1/x$ for $x > 0$. The Chebyshev alternation theorem (see Cheney [1966]) states that the best linear Chebyshev approximation of $f(x)$ over an interval $[a, b]$ is uniquely determined. Moreover, the maximum error occurs at three points on the interval $[a, b]$, and since $f(x)$ is convex, two of these points, a and b, are the boundaries of the interval. Therefore α is the slope of the line connect-

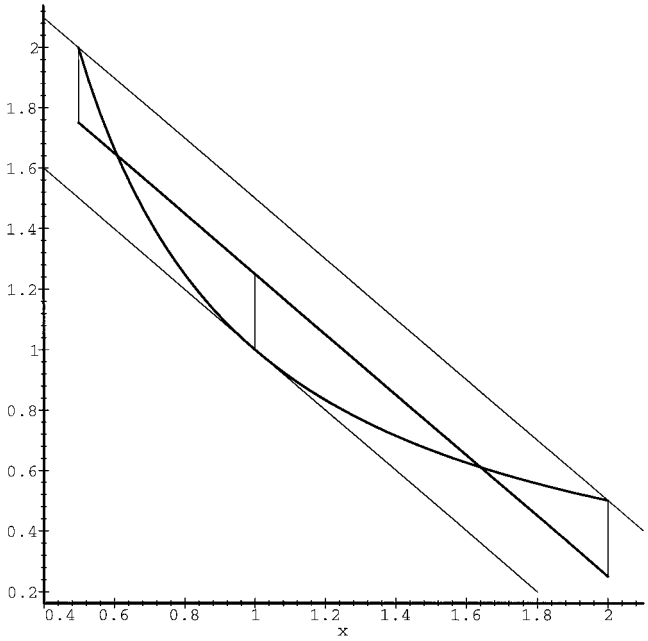

Fig. 9. Linear Chebyshev approximation of $1/x$ over $[0.5, 2]$. The thick line in the center shows the approximation f^{α}, while the thin vertical lines indicate the points of maximum error.

ing $(a, 1/a)$ and $(b, 1/b)$: $\alpha = -1/(ab)$. The third point x' of maximal error is the point on f where the tangent is parallel to line $x' = 1/\sqrt{-\alpha}$.

Finally, first degree Chebyshev approximation is the line with slope α centered between the first line and the tangent (see Figure 9). Thus, with $\beta_1 := 1/a - \alpha a$ and $\beta_2 := 1/x' - \alpha x'$, the value of β is $\beta = (\beta_1 + \beta_2)/2$, and the maximum error on $[a, b]$ is $\delta = |\beta_1 - \beta_2|/2$. Thus the best affine approximation for $1/x$ is

$$\hat{z} = \alpha\hat{x} + \beta + \delta\epsilon_k = (\alpha x_0 + \beta) + \alpha x_1\epsilon_1 + \cdots + \alpha x_n\epsilon_n + \delta\epsilon_k.$$

Linear approximations for other functions can be derived easily using similar techniques.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Michael McCool from the University of Waterloo for their valuable comments.

REFERENCES

ALIAS/WAVEFRONT. 1996. *OpenAlias Manual.*

CHENEY, E. W. 1966. *Introduction to Approximation Theory.* International series in pure and applied mathematics. McGraw-Hill, New York.

COHEN, M. F. AND WALLACE, J. R. 1993. *Radiosity and Realistic Image Synthesis.* Academic Press, 1993.

COMBA, J. L. D. AND STOLFI, J. 1993. Affine arithmetic and its applications to computer graphics. *Anais do VII Sibgrapi,* 9–18. Available at http://www.dcc.unicamp.br/stolfi/EXPORT/papers/affine-arith.

EBERT, D., MUSGRAVE, K., PEACHEY, D., PERLIN, K., AND WORLEY. 1994. *Texturing and Modeling: A Procedural Approach.* Academic Press.

FIGUEIREDO, L. H. 1996. Surface intersection using affine arithmetic. *Graph. Interface '96,* 168–175.

FIGUEIREDO, L. H. AND STOLFI, J. 1996. Adaptive enumeration of implicit surfaces with affine arithmetic. *Comput. Graph. Forum 15,* 5, 287–296.

GREENE, N. AND KASS, M. 1994. Error-bounded antialiased rendering of complex environments. *Comput. Graph.* (*SIGGRAPH '94 Proceedings*, July 1994), 59–66.

GUENTER, B., KNOBLOCK, T. B., AND RUF, E. 1995. Specializing shaders. *Comput. Graph. (ACM SIGGRAPH '95 Proceedings),* 343–350.

HANRAHAN, P. AND SALZMAN, D. 1989. A rapid hierarchical radiosity algorithm for unoccluded environments. In *Proceedings of the Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics.*

HANRAHAN, P. AND LAWSON, J. 1990. A language for shading and lighting calculations. *Comput. Graph. (ACM SIGGRAPH '90 Proceedings),* 289–298.

HEIDRICH, W. 1997. A compilation of affine approximations for math library functions. Tech. Rep. Univ. of Erlangen Computer Graphics Group, in preparation.

KALOS, M. H. AND WHITLOCK, P. A. 1986. *Monte Carlo Methods.* Wiley, New York.

KASS, M. 1992. CONDOR: Constraint-based dataflow. *Comput. Graph. (ACM SIGGRAPH '92 Proceedings),* 321–330.

LEWIS, J.-P. 1989. Algorithms for solid noise synthesis. *Comput. Graph. (ACM SIGGRAPH '89 Proceedings),* 263–270.

LISCHINSKI, D., SMITS, B., AND GREENBERG, D. P. 1994. Bounds and error estimates for radiosity. *Comput. Graph. (ACM SIGGRAPH '94 Proceedings),* 67–74.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: High-speed rendering using image composition. *Comput. Graph. (ACM SIGGRAPH '92 Proceedings),* 231–240.

MOORE, R. E. 1966. *Interval Analysis.* Prentice Hall, Englewood Cliffs, N.J.

MUSGRAVE, F. K., KOLB, C. E., AND MACE, R. S. 1989. The synthesis and rendering of eroded fractal terrains. *Comput. Graph. (ACM SIGGRAPH '89 Proceedings),* 41–50.

NORTON, A., ROCKWOOD, A. P., AND SKOLMOSKI, P. T. 1982. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *Comput. Graph. (ACM SIGGRAPH '82 Proceedings),* 1–8.

PERLIN, K. 1985. An image synthesizer. *Comput. Graph. (ACM SIGGRAPH '85 Proceedings),* 287–296.

PERLIN, K. AND HOFFERT, E. M. 1989. Hypertexture. *Comput. Graph. (ACM SIGGRAPH '89 Proceedings),* 253–262.

PIXAR. 1989. *The RenderMan Interface.* Pixar, San Rafael, CA.

RALL, L. B. 1981. *Automatic Differentiation, Techniques and Applications.* Lecture notes in computer science 120, Springer Verlag, New York.

SLUSALLEK, P., PFLAUM, T., AND SEIDEL, H.-P. 1994. Implementing RenderMan—practice, problems, and enhancements. *Comput. Graph. Forum* (*EUROGRAPHICS '94 Proceedings*), 443–454.

SLUSALLEK, P., PFLAUM, T., AND SEIDEL, H.-P. 1995. Using procedural RenderMan shaders for global illumination. *Comput. Graph. Forum* (*EUROGRAPHICS '95 Proceedings*), C-311–C-324.

SLUSALLEK, P. AND SEIDEL, H.-P. 1995. Vision: An architecture for global illumination calculations. *IEEE Trans. Visualization Comput. Graph. 1,* 1 (March) 77–96.

SNYDER, J. M. 1992. *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis.* Academic Press.

SNYDER, J. M. 1992. Interval analysis for computer graphics. *Comput. Graph. (ACM SIGGRAPH '92 Proceedings),* 121–130.

STAMMINGER, M., SLUSALLEK, P., AND SEIDEL, H.-P. 1997. Bounded radiosity—illumination on general surfaces and clusters. *Comput. Graph. Forum (EUROGRAPHICS '97 Proceedings).*

UPSTILL, S. *The RenderMan Companion.*. 1990. Addison Wesley, Reading, MA.

WARD, G., AP PROFESSIONAL, 1991. A recursive implementation of the Perlin noise function. In *Graphics Gems II,* J. Arvo, Ed. 396–401.

Received April 1997; revised November 1997; accepted December 1997