Chapter 9

Sampling Procedural Shaders

Wolfgang Heidrich

Real-time Shading:
Sampling Procedural Shaders

Wolfgang Heidrich
The University of British Columbia

Abstract

In interactive or real-time applications, naturally the complexity of tasks that can be performed on the fly is limited. For that reason it is likely that even with the current rate of development in graphics hardware, the more complex shaders will not be feasible in this kind of application for some time to come.

One solution to this problem seems to be a precomputation approach, where the procedural shader is evaluated (sampled), and the resulting values are stored in texture maps, which can then be applied in interactive rendering. A closer look, however, reveals several technical difficulties with this approach. These will be discussed in this section, and hints towards possible solutions will be given.

1 Introduction and Problem Statement

In order to come up with an approach for sampling procedural shaders, we first have to determine which aspects of the shading system we would like to alter in the interactive application.

For example, we can evaluate the shader for a number of surface locations with fixed illumination (all light source positions and parameters are fixed), and a fixed camera position. This is the mode in which a normal procedural shading system would evaluate the shader for a surface in any given frame. If the shader is applied to a parametric surface $F(u, v)$, then we can evaluate the shader at a number of discrete points (u, v), and store the resulting color values in a texture map.

In an interactive application, however, this particular example is of limited use since both the viewer and the illumination is fixed. As a result the texture can only be used for exactly one frame, unless the material is completely diffuse. In a more interesting scenario, only the illumination is fixed, but the camera is free to move around in the scene. In this case, the shader needs to be evaluated for many different reference viewing positions, and during realtime rendering the texture for any given viewing direction can be interpolated from the reference images. This four-dimensional data structure (2 dimensions for u and v, and 2 dimensions for the camera position) is called a light field, and is briefly described in Section 4.

If we want to go one step further, and keep the illumination flexible as well, we end up with a even higher dimensional data structure. There are several ways to do this, but one of the more promising is probably the use of a *space-variant BRDF*, i.e. a reflection model whose parameters can change over a surface. This yields an approach with a six-dimensional data structure that will be outlined in Section 5.

No matter which of these approaches is to be taken, there are some issues that have to be resolved for all of them. One of them is the choice of an appropriate resolution for the sampling process. The best resolution depends on many different factors, some of which depend on the system (i.e. the amount of memory available, or the range of viewing distances under which the shaded object will be seen), and some of which depend on the shader (i.e. the amount of detail generated by the shader).

In the case of a 2D texture with fixed camera and lighting, a sample resolution can still be chosen relatively easily, for example, by letting the user make a decision. With complex view-dependent effects this is much harder because it is hard to determine appropriate resolutions for sampling specular highlights whose sharpness may vary over the surface. An automatic method for estimating the resolution would be highly desirable.

Another problem is the sheer number of samples that we may have to acquire. For example, to sample a shader as a space variant BRDF with a resolution of

$256x256$ for the surface parameters u and v, as well as 32^2 samples for both the light direction and the viewing direction requires over 68 billion samples, which is unfeasible both in terms of memory consumption and the time required to acquire these samples. On the other hand, it is to be expected that the shader function is relatively smooth, with the high-frequency detail localized in certain combinations of viewing and light directions (specular highlights, for example). Thus, a hierarchical sampling scheme is desirable which allows us to refine the sampling in areas that are more complex without having to do a high-density sampling in all areas. At the same time the hierarchical method should make sure we do not miss out on any important features. Such an approach is described in the next section.

2 Area Sampling of Procedural Shaders

In this section we introduce the concept of *area sampling* a procedural shader using a a certain kind of arithmetic that replaces the standard floating point arithmetic. This *affine arithmetic* allows us to evaluate a shader over a whole area, yielding an upper and a lower bound for all the values that the shader takes on over this area. This bound can then be used hierarchically to refine the sampling in areas in which the upper and the lower bound are far apart (i.e. areas with a lot of detail). The full details of the method can be found in [10].

We will discuss the general approach in terms of sampling a 2D texture by evaluating a shader with a fixed camera position and illumination. The same methods can however be applied to adaptively adjust the sampling rates for camera and light position.

2.1 Affine Arithmetic

Affine arithmetic (AA), first introduced in [4], is an extension of interval arithmetic [16]. It has been successfully applied to several problems for which interval arithmetic had been used before [17, 20, 21]. This includes reliable intersection tests of rays with implicit surfaces, and recursive enumerations of implicit surfaces in quad-tree like structures [5, 6].

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 [4], [5], and [6].

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 interval $\mathbf{U} := [-1, 1]$. 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 is a matter 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

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

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

3 Non-Affine 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 $z = f(x)$:

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

with $k \notin \{1, \ldots, n\}$. 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 [4] 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 $u = \sum_{i=1}^{n} |x_i|$ and $v = \sum_{i=1}^{n} |y_i|$. 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 iso-surfaces 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 iso-surfaces 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 iso-surfaces 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 [3]. For a more detaile discussion, see [10].

Most multivariate functions can 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 above scheme.

3.1 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 will restrict ourselves to the functionality of the RenderMan shading language [9, 18, 22], 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 [2] and [15]), the described 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 easily be approximated by affine forms using techniques similar to the ones outlined in the previous section. Examples for this kind of domain specific functions 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. For an in-depth discussion of these functions we refer the reader to the original paper [10].

New data types in the RenderMan shading language are points and color values, both simply being 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 parameter 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 will actually be real values on which the affine forms of u and v depend: $\hat{u} = u_0 + du \cdot \epsilon_1$ and $\hat{v} = v_0 + dv \cdot \epsilon_2$.

With this information, we can set up a hierarchical sampling scheme as follows. The shader is first evaluated over the whole parameter domain ($u = 0.5 + 0.5 \cdot \epsilon_1$, $v = 0.5 + 0.5 \cdot \epsilon_2$). If the resulting upper and lower bound of the shader are too different, the parameter domain is hierarchically subdivided into four regions, and area samples for these regions are computed. The re-

cursion stops when the difference between upper and lower bound (error) is below a certain limit, or if the maximum subdivision level is reached. Results of this approach together with an error plot are given in Figure 1.

3.2 Analysis

In our description we uses 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 to not miss any small detail, while tight bounds reduce the number of subdivisions, and therefore save both computation 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 2 shows the wood shader sampled at a resolution of 512×512. The error plots show that interval arithmetic yields errors up to 50% 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.

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

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 get, the more dependencies between the sources of error exist, and the bigger the advantage of AA. These results are consistent with prior studies published in [4], [5], and [6].

The bounds of both affine and interval arithmetic 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.

This leaves us with the method presented here as 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 1 shows a comparison of these numbers in terms of floating point operations (FLOPS) and execution time (on a 100MHz R4000 Indigo) for the various shaders used throughout this paper.

For more complicated shaders the relative performance of AA decreases, since more error variables are introduced due to the increased amount of non-affine operations. The table 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. The density of 0.5 means that 75 percent of the shader will be opaque, while 25 percent will be 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.

For the example of using area samples as a subdivision criterion in hierarchical radiosity, this means that a wall covered with the screen shader would have 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 if only coarse estimates of the error bounds are desired.

4 Light Fields

Let us now consider how the method can be used in a scenario with a varying camera location, but fixed illumination. This is somewhat speculative, because it has never actually been tried. It is therefore to be expected that in practical implementations some new issues will arise that will have to be resolved in future research.

Before we outline an approach for adaptively acquiring light fields from procedural shaders, we will first review the concept of a light field itself.
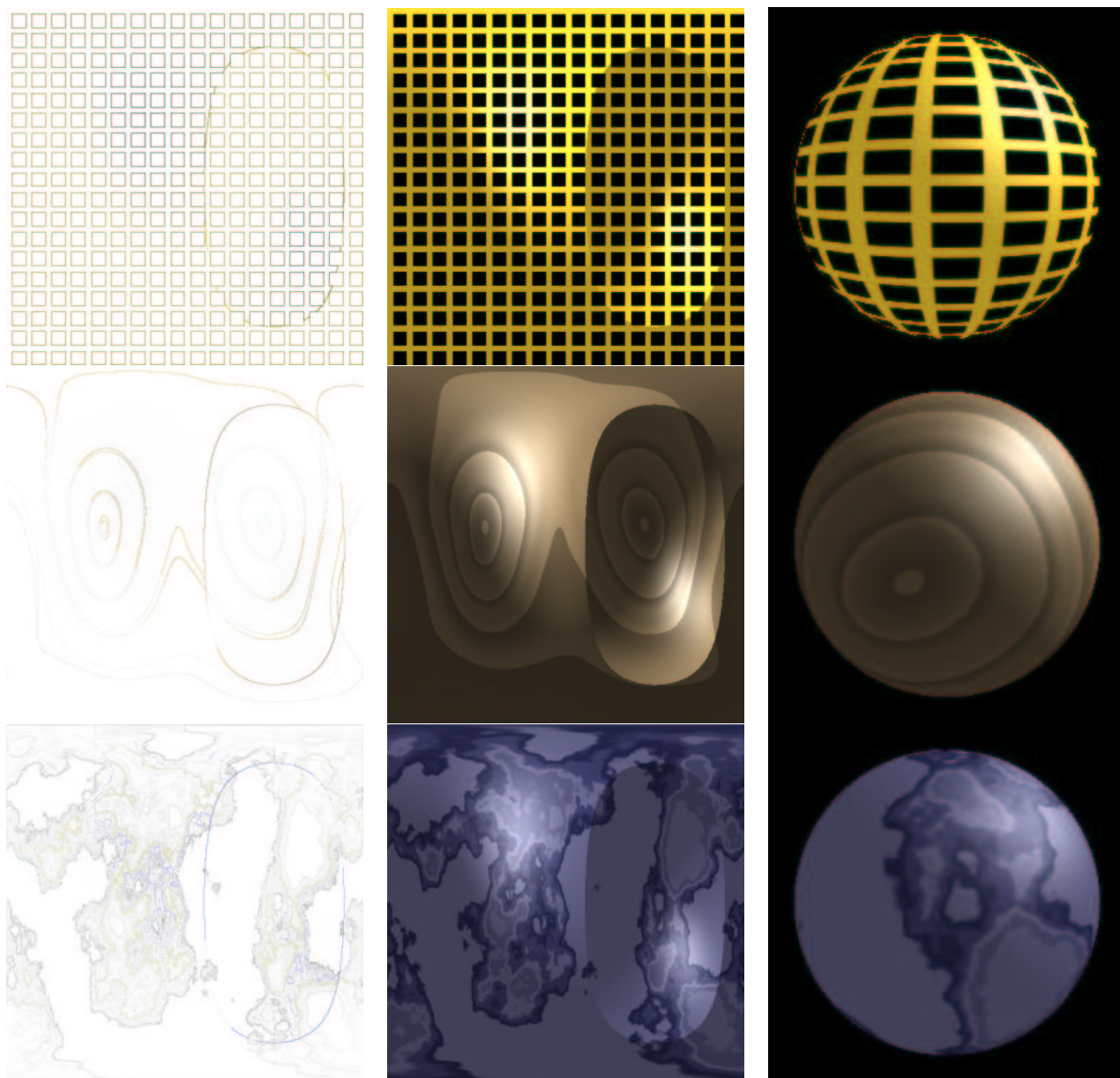
Figure 1: Several examples of RenderManshaders samples with affine arithmetic. Left: per-pixel error bounds, center: generated texture, right: texture applied to 3D geometry.
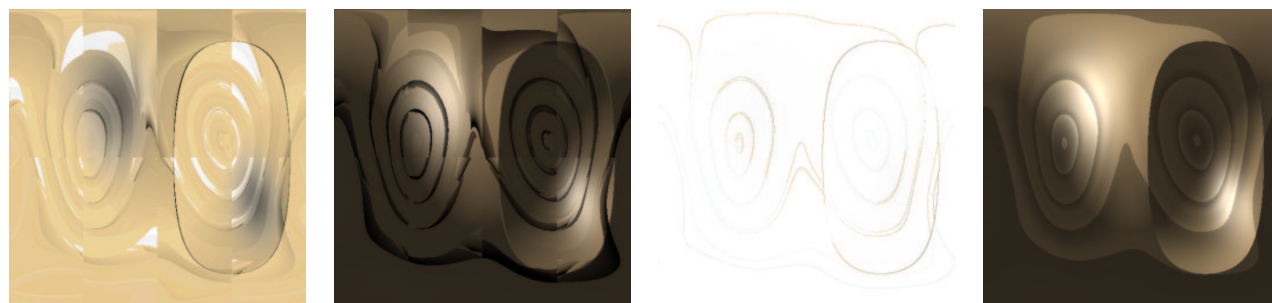

Figure 2: The wood shader sampled at a resolution of 512×512. From left to right: error plot using interval arithmetic, resulting texture, error plot using affine arithmetic, resulting texture.
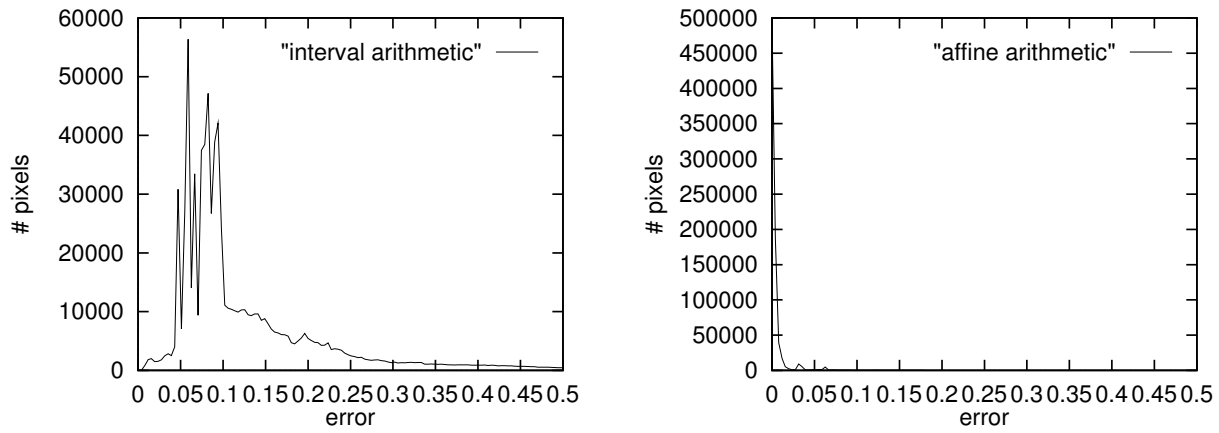
Figure 3: Error histograms for the wood shader for interval arithmetic (left) and affine arithmetic (right).

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

Table 1: FLOPS per sample and timings for 4096 samples, for stochastic point sampling (ps) and AA area sampling (aa).

4.1 Definition

A light field[13] is a 5-dimensional function describing the radiance at every point in space in each direction. It is closely related to the *plenoptic function* introduced by Adelson[1], which in addition to location and orientation also describes the wavelength dependency of light.

In the case of a scene that is only to be viewed from outside a convex hull, it is sufficient to know what radiance leaves each point on the surface of this convex hull in any given direction. Since the space outside the convex is assumed to be empty, and radiance does not change along a ray in empty space, the dimensionality of the light field can be reduced by one, if an appropriate parameterization is found. The so-called two-plane parameterization fulfills this requirement. It represents a ray via its intersection points with two parallel planes. Several of these pairs of planes (also called *slabs*) are required to represent a complete hull of the object. Since each of these points is characterized by two parameters in the plane, this results in a 4-dimensional function that can be densely sampled through a regular grid on each plane (see Figure 4).
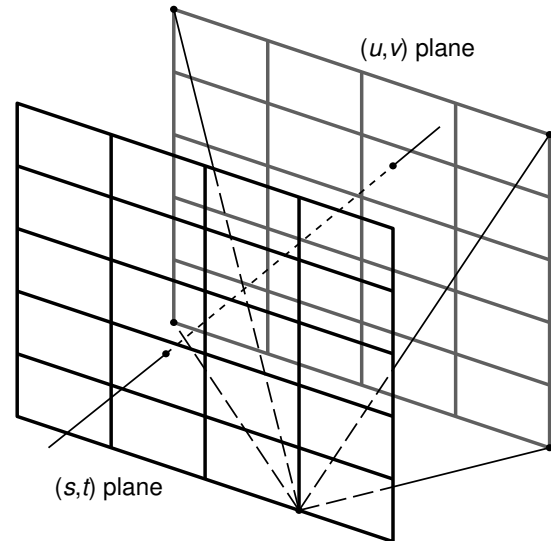
One useful property of the two-plane parameteriza-

Figure 4: A light field is a 2-dimensional array of images taken from a regular grid of eye points on the (s, t)-plane through a window on the (u, v)-plane. The two planes are parallel, and the window is the same for all eye points.

tion is that all the rays passing through a single point on the (s, t)-plane form a perspective image of the scene,

with the (s, t) point being the center of projection. Thus, a light field can be considered a 2-dimensional array of perspective projections with eye points regularly spaced on the (s, t)-plane. Other properties of this parameterization have been discussed in detail by Gu et al.[8].

Since we assume that the sampling is dense, the radiance along an arbitrary ray passing through the two planes can be interpolated from the known radiance values in nearby grid points. Each such ray passes through one of the grid cells on the (s, t)-plane and one on the (u, v)-plane. These are bounded by four grid points on the respective plane, and the radiance from any of the (u, v)-points to any of the (s, t)-points is stored in the data structure. This makes for a total of 16 radiance values, from which the radiance along the ray can be interpolated quadri-linearly. As shown in by Gortler et al[7] and Sloan et al.[19], this algorithm can be considerably sped up by the use of texture mapping hardware. Sloan et al.[19] also propose a generalized version of the two-plane parameterization, in which the eye points can be distributed unevenly on the (s, t)-plane, while the samples on the (u, v)-plane remain on a regular grid.

A related data structure is the *surface light field* [14, 23], in which two of the four parameters of the light field are attached to the surface parameters. That is, u and v correspond to the parameters of a parametric surface, while s and t specify the viewing direction. The details of the different variants of surface light fields are beyond the scope of this document, and we refer the interested reader to the original papers [14, 23].

4.2 Sampling of Light Fields

The sampling method from Section 3.1 can be adapted to the adaptive sampling of light fields from procedural shaders. In addition to computing bounds for the shader over a large parameter domain that we then adaptively refine, we now also compute bounds over a continuum of camera positions. For example, we can start with a large bounding box specifying all possible camera positions, and then adaptively refine it. Or, in the case of a two-plane parameterized light field, we could define the range of camera positions as a rectangular region on the camera plane.

It is not clear at this point how the acquired hierarchical light field can be used directly for rendering in interactive applications. However, a regularly sampled two-plane parameterized light field is easy to generate from the hierarchical one by interpolation. This approach does not resolve the relatively large memory requirements of light fields, but it should dramatically reduce the acquisition time.

5 Space Variant BRDFs

The situation gets even more complex when we also want to allow for changes in the illumination. The most reasonable approach for dealing with this situation seems to be storing a reflection model (BRDF) for every point on the object. That is, instead of precomputing the shader for *all* possible lighting situations (which would require even more space), we only determine the BRDF at every surface location (i.e. a *space-variant BRDF* by considering the effect of a single directional light source which can be pointing at the object from any direction.

As mentioned in the introduction, a space-variant BRDF is a six-dimensional function, and keeping a six-dimensional table is prohibitive in size. Therefore, a different representation has to be found. Again, we should be able to use AAto generate a relatively sparse, adaptive sampling of the shader, which is, however, not well suited for interactive rendering.

On the other hand, the graphics hardware is becoming more and more flexible, so that it is now possible to render certain simple reflection models where the parameters of the model can be varied across the surface [11]. This yields a limited form of space-variant BRDF, where the BRDF actually conforms to a single analytical reflection model, but its parameters can be texture-mapped and can therefore vary across the surface.

Unfortunately, the reflection models considered in [11] are not yet complex enough to capture all the effects that a procedural shader may produce. Other models that provide a general purpose basis for arbitrary effects do exist [12], but it is currently no possible to render them in hardware with space-variant parameters.

Once we have found a reflection model that is expressive enough for our purposes and can be rendered in hardware, we still have to determine its parameters in every point of the object from the hierarchical samples acquired with the adaptive sampling approach. This, again, is an open research problem.

6 Conclusion

In this section we have raised some issues regarding the sampling of complex procedural shaders as a pre-processing step for interactive rendering. We were able to describe a hierarchical sampling scheme that adaptively determines an appropriate sampling resolution for different parts of the shader. The application of this method to determining view-dependent information from a shader in such a way that it is efficient to use in interactive applications is an open problem. We were able to identify some issues arising with this subject, and hinted towards some possible solutions, but more research will have to be done for a complete solution.

References

[1] E. H. Adelson and J. R. Bergen. *Computational Models of Visual Processing*, chapter 1 (The Plenoptic Function and the Elements of Early Vision). MIT Press, Cambridge, MA, 1991.

[2] Alias/Wavefront. *OpenAlias Manual*, 1996.

[3] Elliot W. Cheney. *Introduction to Approximation Theory*. International series in pure and applied mathematics. McGraw-Hill, 1966.

[4] João L. D. Comba and Jorge Stolfi. Affine arithmetic and its applications to computer graphics. In *Anais do VII Sibgrapi*, pages 9–18, 1993.

[5] Luiz Henrique Figueiredo. Surface intersection using affine arithmetic. In *Graphics Interface '96*, pages 168–175, 1996.

[6] Luiz Henrique Figueiredo and Jorge Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15(5):287–296, 1996.

[7] Steven J. Gortler, Radek Grzeszczuk, Richard Szelinski, and Michael F. Cohen. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–54, August 1996.

[8] Xianfeng Gu, Steven J. Gortler, and Michael F. Cohen. Polyhedral geometry and the two-plane parameterization. In *Rendering Techniques '97 (Proceedings of Eurographics Rendering Workshop)*, pages 1–12, June 1997.

[9] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 289–298, August 1990.

[10] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, pages 158–176, 1998.

[11] Jan Kautz and Hans-Peter Seidel. Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000*, pages 51–58, August 2000.

[12] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 117–126, August 1997.

[13] Marc Levoy and Pat Hanrahan. Light field rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, August 1996.

[14] Gavin Miller, Steven Rubin, and Dulce Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, pages 281–292, March 1998.

[15] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 231–240, July 1992.

[16] Ramon E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 1966.

[17] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 41–50, July 1989.

[18] Pixar. *The RenderMan Interface*. Pixar, San Rafael, CA, Sep 1989.

[19] Peter-Pike Sloan, Michael F. Cohen, and Steven J. Gortler. Time critical Lumigraph rendering. In *Symposium on Interactive 3D Graphics*, 1997.

[20] John M. Snyder. *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*. Academic Press, 1992.

[21] John M. Snyder. Interval analysis for computer graphics. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 121–130, July 1992.

[22] Steve Upstill. *The RenderMan Companion*. Addison Wesley, 1990.

[23] D. Wood, D. Azuma, K. Aldinger, B. Curless, T. Duchamp, D. Salesin, and W. Stuetzle. Surface Light Fields for 3D Photography. In *Proceedings of SIGGRAPH 2000*, pages 287–296, July 2000.