Antialiasing Procedural Shaders with Reduction Maps

R. Brooks Van Horn, III and Greg Turk, *Member*, *IEEE*

Abstract—Both texture maps and procedural shaders suffer from rendering artifacts during minification. Unlike texture maps, there exist no good automatic method to antialias procedural shaders. Given a procedural shader for a surface, we present a method that automatically creates an antialiased version of the procedural shader. The new procedural shader maintains the original shader's details but reduces artifacts (aliasing or noise) due to minification. This new algorithm creates a pyramid similar to a MIP-Map in order to represent the shader. Instead of storing per-texel color, pyramid stores weighted sums of reflectance functions, allowing a wider range of effects to be antialiased. The stored reflectance functions are automatically selected based on an analysis of the different reflectances found over the surface. When the rendered surface is viewed at close range, the original shader is used, but as the texture footprint grows, the algorithm gradually replaces the shader's result with an antialiased one.

Index Terms—Antialiasing, texture, procedural shader, surface normal distribution.

1 Introduction

Procedural shaders are widely used in computer graphics today, as demonstrated by the success of shading languages such as the RenderMan Shading Language [1] and GPU shading languages such as Cg and HLSL [2]. Procedural shaders provide several advantages over sampled texture maps, including the ability to show fine detail in close-up views and providing easily tuned parameters during material modeling.

In this paper, we will use the term *procedural shader* to refer to code that is executed during rendering that specifies one or more reflectance functions for a surface. When multiple colors or reflectance functions are used to create spatially varying patterns, sometimes such shaders are also known as *procedural textures*. We will use the terms *image texture* and *texture map* to mean a sampled representation of a spatially varying pattern. Image textures are often stored as 2D arrays of red, green, and blue values, but other surface attributes may also be sampled such as the degree of specularity or the translucence of a surface.

Some of the potential rendering artifacts for procedural shaders can be understood from classical results in signal processing. The Nyquist rate f_N for a continuous function (signal) is two times the highest frequency that is present in the signal. If we fix the viewpoint and the lights in a scene, then a procedural shader represents a continuous function of outgoing radiance from a surface. Although any particular image texture is band-limited, a procedural shader can represent a function that has arbitrarily high frequencies. During image synthesis, the renderer samples the continuous function at discrete positions and thus forms a sampled version of this function. If the samples are regularly spaced and of frequency f_S, then the Shannon Sampling Theorem states that the continuous function can be exactly reconstructed only if $f_S > f_N$, where f_N is the Nyquist rate for the signal. If the sampling frequency falls below this rate, then high frequencies in the signal can masquerade as lower frequencies in a reconstructed signal, and this phenomenon is known as *aliasing*. Some of the more obvious effects of aliasing include stair-stepping at light/dark boundaries and moire patterns when viewing regular textures such as stripes.

Several strategies may be taken in order to combat aliasing. One approach is to increase the sampling rate during rendering, a technique known as *supersampling*. Supersampling does not eliminate aliasing in general but pushes the aliasing to higher frequencies. Supersampling of procedural shaders comes at a high cost because it requires many more compute cycles to calculate the extra samples. Another approach is to move the samples to irregular positions. There are many variants of this, but perhaps, the most popular is *jittering*, where the sample positions are moved by adding small random values. Jittering does not eliminate all problems but converts aliasing artifacts to noise. The human visual system is less sensitive to noise than it is to aliasing artifacts, so this trade-off is usually preferable. Most of the results that we show incorporate jittered sampling. Nevertheless, another strategy is to create a sampled version of the signal prior to rendering and to prefilter this sampled data. An image pyramid (or MIP-Map [3]) is such a prefiltered representation, and our own approach also makes use of this technique.

The task of antialiasing a procedural shader can be extremely difficult even when done by hand. Often, the programmer's time to create a procedural shader is less than the time to create the antialiasing code for the same texture [4]. For some procedural shaders and scenes, an arbitrarily wide range of colors might be mapped to a single

• *The authors are with the Georgia Institute of Technology, 85 5th St. NW, Atlanta, GA 30332-0760.*
E-mail: brooks.van.horn@gmail.com, turk@cc.gatech.edu.

Manuscript received 25 Aug. 2006; revised 1 Feb. 2007; accepted 7 May 2007; published online 9 Aug. 2007.
Recommended for acceptance by J. Dorsey.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0139-0806.
Digital Object Identifier no. 10.1109/TVCG.2007.70435.

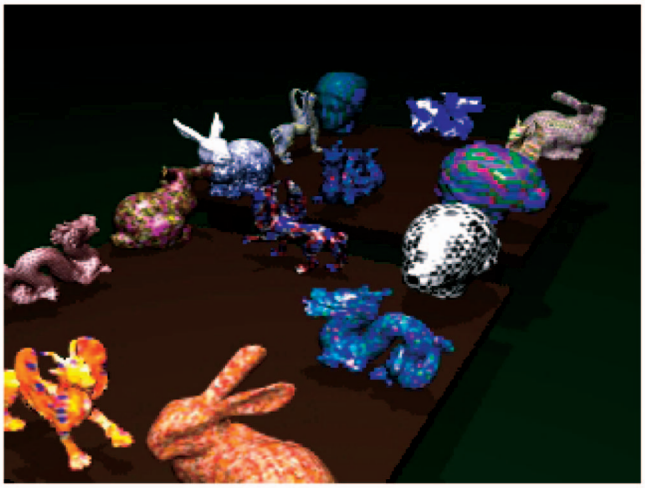

Fig. 1. A scene with multiple procedural shaders that was rendered using procedural reduction maps. The 16-samples-per-pixel version of this animation (which aliases) took more than 10 hours to render, whereas the new method required less than three hours with no artifacts due to minification.

pixel. In order to compute an accurate reconstruction of the pixel, all colors within the *texture footprint* (the texture covered by a subpixel) must be weighted properly and averaged. This cost can be prohibitive for a procedural texture, requiring summing many samples [2]. If done by hand, the best way is to analytically integrate the texture over the footprint. However, this is often difficult or impossible in many cases.

One method of automatically antialiasing procedural shaders is to render the texture at a very high resolution and use the resulting image for an image texture of the object [5]. Although this is a good first step, there are several problems with this method, including the loss of magnification information and loss of spatially varying reflectance properties.

We present a method that automatically reduces and/or eliminates minification artifacts (noise/aliasing) in procedural shaders while maintaining the magnification benefits of procedural shaders. We create a *procedural reduction map* that reduces minification aliasing artifacts, as seen in Fig. 1. The procedural reduction map is a pyramid, similar to a MIP-Map, where each texel in the pyramid corresponds to a region of the textured object. Each texel in the base level of the pyramid stores two pieces of information: the surface normal and a representation of a reflectance function. Specifically, the reflectance function at each texel is stored as a small number of weights that are applied to a set of *basis reflectance functions*. The basis reflectance functions for a given procedural texture are found by automatically analyzing many samples of the texture under a variety of viewing and lighting directions. Texels at higher levels of the pyramid contain basis function weights and *normal distributions*, which describe the variation in surface normals over a portion of the object's surface.

There are three major contributions of our work:

- Our method is automatic given a procedural shader on an object using a minimal set of assumptions. Previous methods specify the way the texture is written or are not automatic.
- We minimize the aliasing or noise artifacts of procedural shaders due to minification without losing the benefit of magnification details.
- For complex textures, our method more closely resembles the original texture than a simple image texture map by using reflectance functions at each texel in the pyramid.

The assumptions required for our algorithm include the following:

- Procedural shaders cannot sample the scene beyond shadows and lights. As such, mirror and near-mirror textures, sampling of indirect lighting, subsurface scattering, and refraction in translucent textures are not handled.
- Only point light sources can be used in the scene.
- Textures cannot use displacement mapping.
- No anisotropic reflectance functions are handled.
- Reflectance function differences are not handled when the differences lie in specular components well outside of the reflection vector.
- The texture must be time invariant.

2 Previous Work

Although the earliest procedural shaders date back to the late 1970s, the technique of writing code to define a texture really blossomed with the invention of shading languages [6], solid textures [7], [8], and solid noise [8]. The open-ended nature of procedural shaders provided power and flexibility, however, the same flexibility increased the difficulty to automatically antialias the same texture and has therefore been attempted by few researchers.

There are several programmer-oriented approaches to antialiasing procedural shaders, and an overview of many of them can be found in the book by Ebert et al. [9]. One of these techniques is clamping, which was described in [10]. Clamping effectively removes aliasing but is appropriate only for spectrally synthesized textures. Analytically prefiltering the texture is an effective means to remove edge aliasing and can be used as a stepping stone for minification antialiasing, but generally, it is difficult except for simple filters [9]. Numerically integrating the texture is obviously the best choice but is usually the least tractable method—sometimes even impossible [11], [4]. Toward the determination of the error created, Heidrich et al. present a method to provide an estimate of error while shading [12].

Multiple level-of-detail textures have also been mostly programmer oriented. Goldman created a fur texture at multiple levels of detail [13], which is similar to Apodaca and Gritz's method of blending between two or more textures when one of them begins to alias [4]. Extending this idea, Olano et al. created a set of texture building blocks with hard-coded antialiased levels of detail that automatically antialias as the texture is built [2]. Hart et al. present a new model for solid texturing that yields an automatically derived box filter width from the shader parameters [14].

One of the problems with manual techniques is that often the programmer spends as much time antialiasing as

creating the texture itself. A more computer-centric method is to simply sample with higher frequency and to jitter the samples. This, however, will not remove aliasing, but will merely change the frequency at which aliasing occurs [9], [2]. Olano et al. introduced automatic simplification of hardware shaders that target the number of texture accesses, a common problem in current hardware [15].

Shader simplification is the technique of reducing the visual details of a texture by modifying the source code of a shader either automatically or by hand [15], [16]. These techniques are often targeted for graphics hardware but have also been used to reduce aliasing by making a similar shader. Note, however, that these techniques do not preserve the original texture. If used as a level-of-detail scheme, popping can be seen as one level of detail is exchanged for another.

Lawrence et al. factored real-world scene information into a series of inverse shade tree "leaves" for user interaction [17]. Their work is similar in nature to ours, except that ours processes synthetic input scenes. Related is the Material Mapping approach that uses multiple texture maps to tag portions of a surface that are represented as different materials [2, p. 138]. Kautz and Seidel make use of precomputed tables for multiple reflectance models and blend them together on a per-pixel basis [18]. McAllister et al. store per-pixel parameters for the Lafortune BRDF in a texture to render spatially varying reflectances on surfaces [19].

Several methods have been proposed to address the problem of highlight aliasing, especially for bumpy surfaces. In Amanatides' work on cone tracing, he broadens the width of a reflected cone to varying degrees in accordance with the reflectance function at a surface [20]. Becker and Max transition between three representations of bumpy surfaces in order to avoid sampling problems [21]. Malzbender et al. make use of per-texel polynomial functions in order to represent material reflectances [22].

During minification of bump maps, it was pointed out by Fournier that traditional methods of antialiasing them (that is, by averaging the normals) would not work and proposed the idea of creating a pyramid of Phong-like distribution of normals [23]. Olano continued this work to include Gaussian distributions of normals [24]. To combat highlight aliasing, Tan et al. use a mixture of Gaussians model to combine the normal variation across a surface and the microfacet distribution of a reflectance function [25].

A lot of the complexity of the antialiasing problem arises from the use of the original Perlin noise function found in [8]. Some of these problems were solved in a more recent work by Perlin [26]. This work did not, however, address the fundamental problem in antialiasing noise, which is that it is not strictly band limited. Wavelet noise addressed this issue so that noise became band limited [27]. This does solve a subset of the problem of antialiasing procedural textures. Unfortunately, it does not help in the antialiasing of high frequencies that arises when the noise values are subsequently modified, especially by discontinuous functions due to conditional branching.

Image texture maps and procedural shaders are opposite sides of the same coin. There are many trade offs between the two, and these are enumerated well in the book by Apodaca and Gritz [4]. The antialiasing of image texture maps has been extensively investigated [28]. Two of the most prevalent techniques include MIP-Maps [3] (which greatly influenced our work), summed-area tables [29]. Nevertheless, another approach is footprint assembly, which averages multiple samples from along the major axis of the footprint [30]. Footprint assembly has been used to improve filtering for ray tracing [31].

3 Algorithm Overview

This section gives an overview of *procedural reduction maps*. To create the procedural reduction map, the user specifies the procedural shader, the size of the procedural reduction map, and the (u, v) parameterized object that uses the texture. Note that the procedural shader does not need to use the (u, v) texture coordinates—these coordinates are just used to unwrap any procedural shader so that it can be sampled on a regular grid for building the pyramid.

The key to our approach is the assumption that the reflectance at each point on the object can be approximated by a weighted sum of a few *basis reflectance functions*, or b_i. The procedural reduction map consists of two elements: the hierarchy (similar to a MIP-Map) of texels and the set of basis reflectance functions, $B = \{b_1, b_2, \ldots b_k\}$, where k is the number of basis reflectance functions. Each texel that is not in the lowest level of the hierarchy is a weighted sum of texels at the next finer level. Furthermore, every texel contains weights corresponding to the amount that each b_i contributes to that texel's reflectance function. The following steps will generate the basis reflectance functions and the pyramid of texels:

1. Rasterize the surface into texels in (u, v) space and also produce the corresponding sample points on the object's surface.
2. Analyze the reflectance functions at each of these sample points.
3. Revisit the surface sample points to find the per-texel weights of the basis reflectance functions.
4. Aggregate the texels' weights up the pyramid.

Our first task is to create the correspondence between 3D sample points $\mathbf{P_i}$ on the object's surface and the 2D texels T_i at the base of the pyramid. For simple objects such as spheres and cylinders, we use analytic mappings. For polygon meshes, we unfold patches in texture space to form a texture atlas.

The second task is to find a good (numerically speaking) set of basis reflectance functions. For each texel T_i and its corresponding sample point on the object, we sample its three-dimensional reflectance function with a large number of input parameters (incoming and outgoing directions). We then analyze this large collection of sampled reflectance functions (one for each texel) to determine the basis reflectance functions B for the given texture. Once the B has been found, each texel in the lowest level of the hierarchy is decomposed into a weighted sum of each b_i, and these weights are then stored in texel T_i. Once the lowest level texels are filled in, the rest of the pyramid is calculated similar to creating a Gaussian pyramid. Although the analysis for a given texture is costly, this cost is amortized by using the

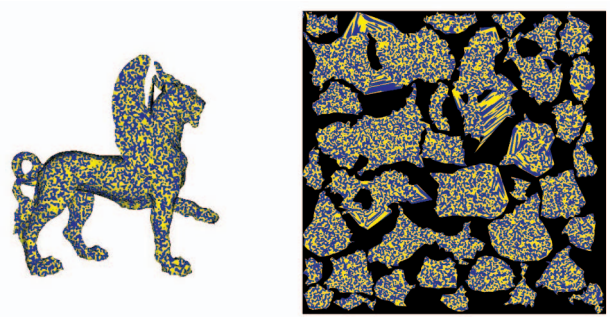

Fig. 2. (a) A procedurally textured mesh is unfolded into patches and (b) its texture is sampled into the base level of the pyramid.

resulting procedural reduction map repeatedly in the many frames of an animated sequence.

When rendering a textured object, a pixel's color is found based on the object's (u, v) coordinate and an estimate of the pixel's texture footprint area. Trilinear interpolation in the pyramid is performed to give weights for each b_i. Each b_i is rendered and weighted accordingly.

4 Creating Procedural Reduction Maps

In this section, we describe in more detail each of the steps for creating the procedural reduction map.

4.1 Texel/Sample Point Correspondence

Our first task is to create the correspondence between 3D sample points on the object's surface and the 2D texels at the base of the pyramid. For simple objects such as spheres and cylinders, this correspondence can be done with an analytic mapping. To build these correspondences for meshes, we follow the work by Sander et al. [32] and Zhang et al. [33]. We assume that the object to be textured has been parameterized into one or more texture patches that can be assembled into one texture atlas. Fig. 2b illustrates the atlas and its patches for the textured object shown in Fig. 2a. There are many automatic methods of creating such a texture atlas for polygon meshes, and we use the method of Zhang et al. [33]. We note that, similar to image texture maps, the better the mapping between UV-space to object-space, the more accurate the antialiasing results.

Once we have created the (u, v) surface parameterization for an object, we then rasterize each mesh polygon to produce (u, v) samples that correspond to the texels at the finest level in the pyramid. We will refer to this collection of texels $T_1, T_2, \ldots, T_n$ for the n texels at the base of the pyramid. Note that if the procedural shader uses displacement mapping, these points will not necessarily be valid, and we therefore constrain the procedural shader to not use displacement mapping. With these texels now in hand, we can analyze the reflectance functions at the corresponding locations on the surface of the textured object.

4.2 BRDF Sampling and Analysis

Our premise for creating procedural reduction maps is that all but the most pathological textures are actually a spatially varying blend of a small number of different reflectance functions. Our task, then, is to recognize a minimum set of k "elementary" reflectance functions (B) that can be combined to form any of the reflectance functions in the texture. B can be any set of reflectance functions so long as all reflectance functions of the texture are within the space spanned by that set. However, we also place two additional constraints on B. First, we want to minimize the number of basis reflectance functions for both speed and storage concerns. Second, we want the vectors storing the sampled basis reflectance functions to be as orthogonal as possible. This means that the basis reflectance functions are as dissimilar as possible in order to avoid numerical inaccuracy problems during processing.

Many textures could be written in such a way as to provide the reflectance function at a specific point, however, it is not necessarily straightforward in all textures to do so. Furthermore, preexisting textures will likely not have been written in such a way. Therefore, for these cases, we wish to recognize the basis reflectance functions in an entirely automatic way. We do this in two steps. First, for each texel, we sample the reflectance function on the object (corresponding to the texel) from a large number of light and view directions. Second, we analyze this collection of sampled reflectance functions by solving a non-negative least squares (NNLS) problem. The result of this analysis is a set of basis reflectance functions. We will use the notation $R(\omega_i, \omega_r)$ to describe a single reflectance function, where ω_i and ω_r are the incoming and outgoing (reflected) light directions, respectively.

Each texel T_i corresponds to some reflectance function R_i. Although each such reflectance function may be defined analytically, we sample a given R_i to create a sampled version of the function. We query the reflectance function using a variety of incoming and reflected directions ω_i and ω_r over the hemisphere given by the object's surface normal. We use the same set of incoming and outgoing directions for sampling each reflectance function R_i. Each color value that is returned is placed in the vector V_i that gives the sampled representation for the reflectance function. Thus, for n texels, there are n sampled reflectance functions V_i.

4.2.1 Sampling Details

In principle, any number of samples of the reflectance function may be used. In practice, however, there is a balance between using few samples for speed and using many samples to accurately represent the reflectance function. We note here that this principle of minimizing the number of samples for reasonable speeds directly led to the assumption mentioned earlier that the texture could not sample the scene beyond the shadows and lights. If the texture were allowed to do that, then the number of samples required to accurately recognize different reflectance functions would increase drastically.

Fig. 3 lists the angles that we use to sample the reflectance function. To minimize the amount of sampling, we use the assumptions that the texture is isotropic, and the specular component (if any) are located along or near the reflection vector. Isotropic reflectance functions can be described using three variables: incoming elevation (θ_i), outgoing elevation (θ_o), and difference in rotation around the pole between the two (ϕ). Note that the case where the outgoing direction is the normal, we have special sampling,

Incoming (θ_i)	Outgoing (θ_o)	Rotation (ϕ)
0	0	0
0	2	0
0	2	90
0	10	0
0	10	90
0	15	0
30, 45, 60, 85	θ_i	180
30, 45, 60, 85	θ_i	180 + 2
30, 45, 60, 85	θ_i	180 + 10
30, 45, 60, 85	θ_i + 2	180
30, 45, 60, 85	θ_i + 10	180
30, 45, 60, 85	15	180
30, 45, 60, 85	75	180
180	0	0

Fig. 3. Angles (in degrees) sampled for isotropic reflectance function matching and determination. The first column is the incoming elevation, the second is the outgoing elevation, and the last column is the rotational difference around the pole between the two. The last sample (bottom row) is used for transparency, and this gives us 35 total reflectance samples.

but for cases where (θ_o) is nonzero, the sampling is the same and thus listed in the figure in the same line.

4.2.2 Analyzing to Find B

Given this collection of sampled reflectance functions $\{V_1, V_2, \ldots, V_n\}$, we wish to analyze them to create the set of basis reflectance functions B. Hopefully, the number of basis reflectance functions is far smaller than n, and each of the reflectance functions R_i can be represented by a weighted sum of each b_i. One possible technique is to use principal component analysis (PCA). Consider, however, the following example. Imagine a texture that is a blend of red and green colors, and the spatial pattern of red versus green may be based on something like a band-limited noise function. Rather than recognizing red and green as the constituent components of the texture, PCA will return the vector yellow (the average of red and green) as being the dominant component. For reasons we will discuss shortly, we seek an analysis technique that will give us basis vectors near the convex hull of the samples V_i rather than in the interior. To perform this analysis, we solve a nonnegative least-squares problem.

Given a large collection of vectors, NNLS finds a set of basis vectors that span the subspace of the input vectors. In addition, the weights required to represent any of the input vectors will all be positive [34]. In the case of the red-green texture example, NNLS will select red and green as the basis vectors. For further insights into various matrix factorization methods, see [17].

As with other analysis tools such as PCA, the computational requirements of NNLS becomes prohibitive for a large number n of input vectors. Since we typically wish to analyze the reflectance functions for a 1,024 $\times$ 1,024 grid of texels, we take a multistep approach to analysis. We break up our texels into smaller blocks, where each block is typically 32 $\times$ 32 in size. We run NNLS for each of these smaller blocks, and we save the top three selected basis reflectance functions from each block. This means that some basis reflectance functions might be lost during this stage. We then aggregate all of these selected reflectance functions and again perform NNLS on this list. The most significant reflectance functions from this factorization become the set of basis reflectance functions, B.

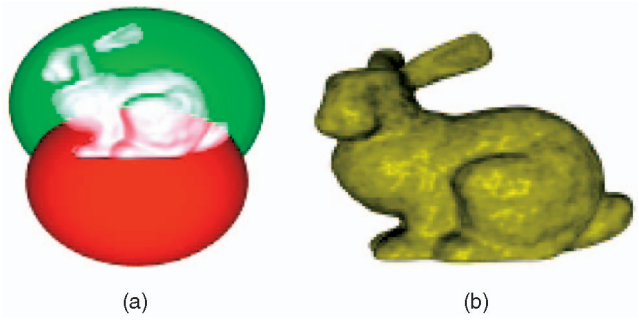

Fig. 4. (a) A procedural reduction map of a texture using transparency. Light rays are not refracted when entering the transparent object. (b) A procedural reduction map rendering of a bump mapped object.

After selecting B, we make an additional pass over the texels to determine their weights. This is the simple task of projecting each reflectance function vector R_i of a texel T_i onto each basis function vector b_i. The resulting k weights from this are stored at each texel. The projection of the vectors into that subspace also identifies vectors that are not in that subspace defined by B. We then add any of these reflectance functions that are outside of the space of reflectance functions defined by the current B by adding these reflectance functions to B. (These are the ones that may have been thrown out during the NNLS processing step.) We note that this rarely happens.

4.2.3 Transparency

Adding transparency requires only a little addition to the current sampling scheme. *Transparency* in this sense means filtered light with no refraction through the object (see Fig. 4a). One set of incoming and outgoing light directions are picked to determine if any light is filtered through the object, as shown in the last row in Fig. 3. If there is a transparent component to the texture, then this result will be nonblack.

4.3 Basis Reflectance Function Representation

As stated previously, each texel in a procedural reduction map contains a weight for each b_i, the weighted sum of which represents the reflectance function for that portion of the textured object. The representation is independent of the procedural reduction map, but we explain our chosen representation and its ramifications here.

Initially, we explored the possibility of using a collection of spherical harmonics to represent a reflectance function. This representation is satisfactory for low-frequency reflectance functions, and it has the advantage of being fairly fast to evaluate. Unfortunately, spherical harmonics have trouble capturing sharp highlights. One potential solution would be to move to a refinable representation of spherical functions such as spherical wavelets [35]. Another possibility is to make use of a factored representation of the BRDF [36], [37].

Because there is to date no fully general approach to creating such factorizations, we chose to take a different approach.

Instead of creating a tabulated version of each basis reflectance function based on a large number of samples, we decided to use the actual procedural shader code itself to represent each b_i. Given a b_i as returned by NNLS, we find the texel T_i whose sampled reflectance function is closest to b_i. We record the corresponding position $\mathbf{P_i}$ on the object's surface, and we also save the surface normal $\mathbf{n_i}$ at this point. This position and normal, together with the procedural shader code, is our b_i. When, during rendering, we need to evaluate a given b_i, we evaluate the reflectance by querying the procedural texture at this point.

Given that our representation of each b_i is the procedural texture itself with specific parameters, the reason for using NNLS for finding the b_i becomes evident. In many instances, PCA and similar analysis methods often produce basis vectors that are nowhere near the original sample vectors. If such a basis vector is selected, there might be no texel that actually has the desired reflectance function. NNLS selects its basis vectors from the actual input vectors that are being analyzed. For our purposes, this means it will yield b_i that can be accurately represented by the reflectance functions of actual points on the surface of the object.

4.4 Filling the Higher Pyramid Levels

There are two parts to filling in the texels in the procedural reduction map: creating the texels at the finest level of the hierarchy and determining the texels in the high levels. Section 4.2 described creating the finest level texels, and in this section, we will discuss how to aggregate the existing texel information up to the other levels of the pyramid.

Similar to MIP-Maps and Gaussian pyramids, each progressively higher level texel is determined by a weighted average of child texels. This procedure is repeated until all levels of the procedural reduction map hierarchy are filled in. Note, however, that some portions of a texture atlas can be empty, that is, may not correspond to any portion of an object. See the black regions in Fig. 2b for an example of such regions. Sanders et al. noted these empty texels can be a problem if the (u, v) coordinates are near a patch boundary [32]. In addition, these empty texels should not contribute to texel values at higher pyramid levels. Their solution is to use the pull-push pyramid algorithm in [38] to bleed the color information into the empty texels. We use the same pull-push technique, but where they applied it to color channels, we apply it to our basis function weights.

4.5 Surface Normals

Surface normal information is needed both during reflectance function analysis and when rendering the texture from a procedural reduction map. The sampling of the reflectance function during analysis requires the surface normal to be known so that the appropriate incoming and outgoing light directions can be used. For many procedural shaders, the *geometric normal*, that is, the normal of the object at that point, is the surface normal used by the procedural shader.

In the case of procedural shaders that perturb the normals during shading calculations, that is, those that have bump maps (Fig. 4b), the system should estimate the perturbed normals by sampling the reflectance function. The new normal after being perturbed by the procedural shader is often called the *shader normal*. However, for ease of discussion, we extend this term to include the geometric normal even when the procedural texture has no bump mapping.

When the shader normal does not equal the geometric normal, sampling of the reflectance functions has to be done carefully. Transformation of the incoming and outgoing light directions need to be applied based on the normal that the texture will use in its calculation. Thus, using the geometric normal when it does not equal the shader normal will result in not recognizing the same reflectance function. We need to automatically determine the shader normal so that we fulfill the goal of minimizing the number of basis reflectance functions.

For textures that obey Helmholtz reciprocity ($R(\omega_i, \omega_r) = R(\omega_r, \omega_i)$), Zickler et al. provide a way of calculating the true surface normal from six samples of the reflectance function [39]. Even though most real-world reflectance functions obey Helmholtz reciprocity, it is possible (and common) for textures to break this law. Therefore, this method does not help in the general case.

For the general case, we use a heuristic to determine the shader normal. Starting from the geometric normal, we adaptively sample along two distinct great arcs toward the opposite point on the sphere. The point that those two arcs first reach the color black creates a plane (along with the center of the sphere) that defines the shader normal. This can be achieved with 15 samples per line (30 total) using a binary search (results are either black or not), giving an estimate of the shader normal that is within 10^{-4} radians of the original answer. This falls near the borders of numerical accuracy and is sufficient for our method. For a given sample point on the sphere, both the incoming and outgoing light sources are placed on that point and the reflectance function sampled. We note that while this method works well for many shaders, it will fail for shaders that return black for nonbackfacing directions.

Alternatively, many procedural shaders invoke a separate function to determine the shader normal. With access to that, the shader normal determination becomes trivial, fast, and exact.

Now that the corresponding surface normal for a texel has been determined, we store the shader normal information at the given texel's location in the finest detail level of the pyramid. When creating the higher levels of the pyramid, we not only average together the basis reflectance function weights from child texels, but we also aggregate surface normal information. In particular, we use the *normal distribution* scheme by Olano and North [24]. Briefly, they describe a distribution of surface normals as a mean normal vector (a 3D vector) and a 3×3 symmetric covariance matrix. The covariance matrix is an approximation of a distribution of surface normals on a sphere. Each Gaussian approximation can be linearly combined by transforming the covariance matrix into the second central moment. Since it is a linear combination, weighted combinations of these distributions can be done. This perfectly fits our needs for

Texture	k	PRM Size (MB)	Ratio
Thresholded Griffin	2	8.4	2.0
Shiny/Dull Scaled Dragon	2	8.4	2.0
Stone Bunny	3	9.8	2.3
Marble Igea	4	11.2	2.7
Bump Mapped Bunny	1	15.4	3.7
Transparent Bunny	1	7.0	1.7

Fig. 5. PRM Size. The second column lists the number (k) of basis reflectance functions in the procedural reduction map. The third column is the size of the PRM, and the fourth column is the ratio of the PRM size to a MIP-Map of the same base size.

combining distributions while creating the higher levels of the pyramid.

A nice feature of this normal distribution representation is that the nine numbers that represent such a distribution can be combined linearly to merge two or more distributions into a single broader distribution. This is how we merge the normal distributions as we move up the pyramid of texels. For flat surfaces, the covariances for the normal distributions will remain small as we go up the pyramid. For curved surfaces, texels from higher levels must represent a substantially curved portion of the surface. In these cases, the normal distributions become broad. The normal distributions also become broad in the case of bump mapping, since higher level texels get their normal distributions from many normals that may cover a substantial portion of the Gauss sphere.

We will return to these normal distributions when we discuss rendering from a procedural reduction map.

4.6 Memory Usage

The per-texel cost of a procedural reduction map is one byte per basis function weight and four bytes to represent the direction of the average normal (two bytes each for angles ϕ and θ). All procedural reduction maps used in this paper were created at 1,024 $\times$ 1,024 resolution. For models with three basis reflectance functions, which is typical, this results in 9.8 Mbytes per texture (factoring in the 4/3 cost of the pyramid). A standard MIP-Map texture using three bytes per texel for color requires 4.2 Mbytes of storage at the same resolution. For objects that have considerable variations in surface normals, we also store a normal distribution map per-texel, and this requires one byte for each of the six unique elements in the matrix. This raises the memory usage to 18.2 Mbytes, and this is why the bump-mapped bunny in Fig. 5 has such high memory usage. These memory costs are per object, similar to any other use of texture memory for a given model.

5 Rendering

Procedural reduction maps can be used with either a scan-conversion or a ray tracing renderer. Later, we will present rendering results for both a ray tracer running on the CPU and for scan-conversion GPU rendering. During rendering, the procedural reduction map is called with the (u, v) coordinate and an estimate of the area of the *texture footprint* (the portion of the texture visible in the current pixel). We will return to the computation of the footprint later in this section. The texture footprint gives all necessary information needed to perform trilinear interpolation from the pyramid. The resulting interpolation can be thought of as a new texel, T_f.

Given T_f, we have weights for each b_i, and we have the normal distribution. The final radiance will be computed in one of two ways. We will refer to these methods as the *normal approximation* and the *multiple samples approximation* methods.

5.1 Normal Approximation

For fast rendering, we simply use the mean of the normal distribution as the surface normal $\mathbf{n}$ for shading, and this is the only method that we use for GPU rendering. We examine the weights for each b_i and evaluate the reflectance functions with nonzero weights. As described in Section 4.3, each b_i is represented implicitly as a normal $\mathbf{n_i}$ and a position $\mathbf{P_i}$ on the object that has the appropriate reflectance function associated with it. We find the transformation that rotates $\mathbf{n}$ to be coincident with $\mathbf{n_i}$, and then use this transformation to rotate the light and the view vectors into the appropriate coordinate system. We then invoke the shader at the location $\mathbf{P_i}$ with these transformed light and view vectors. The returned value from the shader is the outgoing radiance.

We note here that rendering according to the normal approximation method is often not entirely accurate to the original reflectance distribution. (Note that this problem is different than having two widely different reflectance functions next to each other in the textured object, which is handled by the procedural reduction map algorithm.) If the reflectance function changes drastically over the normal distribution in T_f (that is, the specular exponent and coefficient is high), then aliasing can occur. For this reason, our system will in some cases average together multiple samples.

5.2 Multiple Samples Approximation

For cases where the normal approximation method fails, we resort to adaptive supersampling in screen space. Using the original rendering program, we supersample this pixel. The number of supersamples can be determined in a variety of ways, but for our examples, we used 16 since this is what we are comparing against. We only use this method for our CPU ray tracing renderer, since this method requires a different numbers of samples per pixel.

Note that when this case arises, it is exactly the same as the original automatic antialiasing algorithm of supersampling. With the exception of a very small overhead involved in the procedural reduction map algorithm, the costs are the same.

Fortunately, this supersampling case does not occur often. Among all procedural textures used in this paper and the accompanying video, only the highly specular highly curved bump-mapped bunny required the multiple samples approximation. Furthermore, even this case required the multiple samples approximation only a small fraction of time (see Fig. 6). In the animation sequence of the bump-mapped bunny, an average of only 1.6 samples per pixel was required. This number is small because the specular highlights only occur in certain areas, and those can be

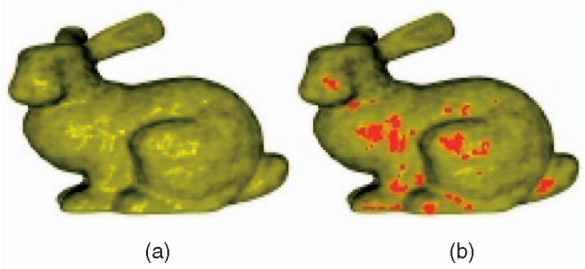

Fig. 6. Frequency of the multiple samples approximation. (a) Procedural reduction map, 1 sample per pixel (except for multiple samples regions). (b) Multiple samples regions are marked with red. During the animation, an average of 1.6 samples per pixel were used.

determined by comparing the reflection vector with respect to the normal distribution.[1] Noting that we have the directions of the incoming and outgoing light directions, and the half-vector between them is the normal that would cause the specular component to become a factor. Therefore, we need to see if the half-vector is within the distribution of normals. To do this, we compare the half-vector to the Gaussian distribution of normals and determine how many standard deviations away it is from the center. To calculate this, we invert the covariance matrix of the normal distribution for texel T_f and multiply by the half-vector after it has been adjusted for the center of the distribution. If it is farther than 2.5 standard deviations, then we use the normal approximation method instead.

Although there are many rendering algorithms that make use of adaptive supersampling, most of them are based on color differences between samples. Because our approach is based on differences in surface normals, it should avoid the problems from bias that arise when performing adaptive sampling based on color.

The final observation of this method is that, despite intuition, our animations alias less than or equal to supersampling at every pixel. Obviously, in pixel regions where the screen-space approximation method is used, then there is an equal amount of aliasing. Therefore, the interesting comparison occurs in the regions where the normal approximation method is used versus supersampling. The normal approximation does not involve the specular component, even though it is possible that there is a small portion of the normal distribution that does include it. From frame to frame, the color changes gradually due to the nature of the pyramid of averaged normals. On the other hand, supersampling does not have access to such information. When a specular region takes up only a small fraction of a pixel, supersampling may by chance occur for a disproportionately large number of the samples. Since the specular component typically has a much higher intensity, this sudden increase in color intensity that will disappear in the next frame is seen as aliasing.

The frequency of this approximation can be seen in Fig. 6.

5.3 Calculating the Texture Footprint

For our ray tracing renderer, we estimate the texture footprint area by shooting rays at the four corners of a pixel. When all four rays hit the same textured surface, the footprint area calculation is straightforward. Each ray/object intersection corresponds to a position in texture space, and we enclose the four given positions in a square that becomes our footprint. According to the usual MIP-Map level calculation [3], we then perform a lookup in our image pyramid. Although we have not yet done so, our method could easily be extended to use footprint assembly in order to perform anisotropic filtering [30].

In the event that two or more objects are hit during ray/object intersection, we shoot more rays that further subdivide the pixel. We then use these additional rays to estimate footprint sizes on the various objects and, in addition, to determine what fraction of the pixel's color will be due to each object. This approach generates smoothly antialiased silhouettes of textured objects. Textured mesh objects that have been unfolded into multiple patches in an atlas are also specially treated. If the four original rays for a pixel hit more than one texture patch in the same atlas, there is a danger of miscalculating the footprint size. For this reason, rays that strike different patches are treated just as though they hit separate objects, causing additional rays to be shot that then give the footprint areas and the contributing weights for the various patches.

Note that the finest level of the image pyramid represents the finest details that the procedural reduction map can antialias. If the area of the texture footprint is less than the size of the finest texels, we do not use the procedural reduction map and instead invoke the original shader code. If the texture footprint is nearly the same as the smallest texel size, we perform a weighted blend (based on footprint area) between the value given by the procedural reduction map and the value given by the shader code. This creates a smooth transition between the procedural reduction map and the original shader.

6 Results for CPU Ray Tracing

We implemented rendering using our procedural reduction maps with both a CPU ray tracer and with GPU scan-conversion. For the ray tracer, we tested our approach on a single-processor Intel Pentium 4, 2.8-GHz machine with 2 Gbytes of memory. The renderer we used was a modified version of POV-Ray. We chose to use a public-domain renderer as our software platform so that during rendering, we could easily catch and modify procedural shader calls to use procedural reduction maps. We do not see any conceptual barrier to using procedural reduction maps in other CPU renderers as well.

Fig. 7 demonstrates several procedural shaders of varying types that are common in computer graphics: thresholded noise, a regular (hexagonal) pattern with differing specular components, cellular texture, and marble. The figure demonstrates the textures at varying levels of zoom (each perfectly antialiased). The figure also demonstrates what kind of aliasing can occur when nothing is done (1 sample per pixel, no hand antialiasing), supersampling is performed (16 samples per pixel), and when the procedural reduction map is used with only 1 sample per pixel. For the aliasing demonstrations, the object is far away, and then, the resulting image is magnified.

1. This is valid due to our assumption that the specular highlights are only found along the reflection vector.

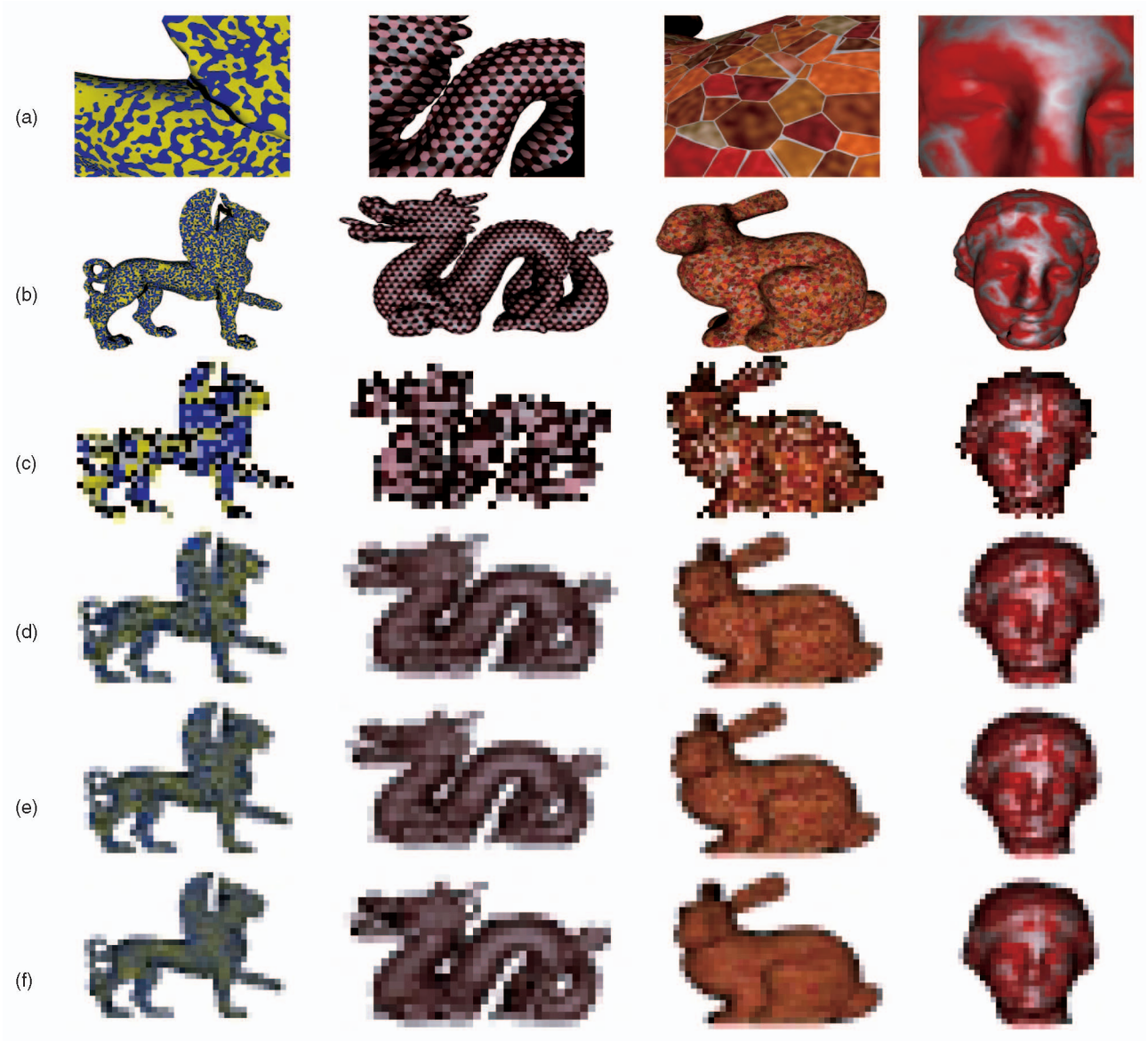

Fig. 7. Four representative textures are shown. The first two rows demonstrate the texture with very high-quality renderings. (a) The texture when magnified. (b) The texture on the entire object when the object is close enough such that there is no aliasing. For the next four rows, the object is placed far from the viewing screen and then rendered. The resulting image is then enlarged without interpolation. (c) One sample per pixel rendering. (d) Sixteen samples per pixel rendering. (e) One hundred samples per pixel rendering, which is close to the ideal image. (f) The procedural reduction map with only one sample per pixel rendering.

From left to right in Fig. 7, each texture demonstrates a common and difficult aspect of automatic antialiasing of procedural shaders. First, the thresholded noise texture applied to the griffin model (10K triangles). This is difficult to antialias due to the discontinuous function at the boundary between blue and yellow. Although wavelet noise [27] has greatly improved the antialiasing of noise, it does not help in this situation.

The second texture is the hexagonal pattern applied to the dragon model (20K triangles). This pattern has three different reflectance functions, but what makes it most interesting is that one of these has a much different specular component. Traditional MIP-Maps cannot handle this type of texture, and this type of texture is often left as is and supersampled for antialiasing purposes.

The third texture is a cellular (Worley) texture applied to the bunny model (10K triangles). Cellular textures are popular, but they can quickly become difficult to antialias. Antialiasing this texture by hand would be difficult at best and trying to make this a traditional MIP-Map loses the magnification benefits of the original texture.

The fourth texture is another common form of procedural shader, the solid texture marble, applied to the igea head (10K triangles). The white regions of this texture have been made more glossy than the dark portions. The difficulty with this type of texture is the nonlinear correspondence between the noise function and the color space [9], as well as the blending of a glossy component into portions of the texture.

We refer the reader to the accompanying video clips to see the quality of these results for animated sequences. The

Texture	Texture Calls (PRM)	Texture Calls (16)	1-Sample	16-Samples	PRM (1-Sample)
Griffin	6,392.7	39,868.6	5:01	13:16	6:20
Dragon	6,295.1	40,115.3	8:25	16:16	9:33
Stone Bunny	12,708.3	63,171.2	5:25	16:29	6:35
Bump Mapping	6,282.6	63,172.5	5:32	11:41	6:08
Igea	16,269.4	66,388.4	5:36	16:47	7:09

Fig. 8. Rendering costs for each of the 300 frame sequences. The average number of texture calls per frame in the procedural reduction map is listed in "Texture Calls (PRM)." The average number of texture calls per pixel in the 16 samples per pixel case is listed in "Texture Calls (16)." The timings for rendering the associated texture in the rotation animation is listed for "1-Sample" per pixel, "16-Samples" per pixel, and the "PRM (1 sample per pixel)." Note that all models are have 10K triangles except the dragon, which is has 20K triangles. Rendering times are in (minutes:seconds).

Model	k	No. Triangles	1-Sample (sec.)	16-Samples (sec.)	PRM (sec.)	Algorithm (sec.)	Ratio (1-S / PRM)
Igea	2	10,000	15.6	117.7	28.6	13.0	0.545
Igea	4	10,000	15.6	117.7	28.8	13.2	0.542
Dragon	2	20,000	24.1	186.6	42.1	18.0	0.572
Dragon	4	20,000	24.1	186.6	42.5	18.4	0.567

Fig. 9. Cost of ray-object intersection tests and the procedural reduction map algorithm. Each row is a model with either two or four basis reflectance functions. The costs of texture computations have effectively been eliminated. Timings are for rendering 1,000,000 pixels. The second to last column is the additional cost of running the procedural reduction map algorithm. The last column is ratio of 1-Sample over PRM and demonstrates the algorithm runs at a little over half the speed of 1-Sample. The PRM for these examples does not make use of the normal distribution map.

differences are even more pronounced in animated scenes. In particular, the 16 samples per pixel videos have noticeable aliasing and/or noise artifacts. We also note here that this algorithm does not handle silhouette aliasing but rather minification aliasing.

Fig. 10 gives timing results for the procedural reduction map creation, along with timings for various parts of the creation portion of the algorithm. Only one of our test models, the bumpy bunny, required automatic surface normal determination, and this required one hour and 45 minutes to compute. Although the NNLS portion of the algorithm is by far the most expensive aspect of the algorithm, note that for many textures, such as the cellular texture, that finding B is difficult even for humans. We note that although the offline texture creation times are long, the benefit is greatly reduced final rendering time.

Fig. 8 gives the rendering timing results for several textures, some with a corresponding procedural reduction map and some without. The two main costs in rendering are the number of rays cast and the number of texture calls. The table attempts to delineate the costs and benefits of our algorithm. In the animation, notice that visually, the procedural reduction map is of the same quality as 16 samples per pixel (and is sometimes better). Therefore, it is often a savings in rays cast since the algorithm requires only one ray per pixel. However, the table also lists the number of texture calls. For many surfaces and associated textures, our algorithm has superior performance in the number of texture calls. Notice that our PRM technique is faster than 16 samples by a factor of 2 to 3 and produces comparable or higher quality results (see video).

Texture	Sampling	NNLS	Filling	Total
Griffin	0:01:16.1	0:13:45.2	0:02:09.3	0:17:52
Dragon	0:01:17.7	7:37:53.4	0:02:05.6	7:41:58
Bunny	0:02:36.1	7:03:19.4	0:02:20.0	7:08:58
Igea	0:02:20.9	6:31:23.6	0:03:00.4	6:37:06
Bump Map	0:01:48.3	0:03:44.6	0:02:17.0	0:09:52
Transparency	0:00:43.5	0:03:41.2	0:02:20.1	0:07:02

Fig. 10. Timings (hours:minutes:seconds) for creating procedural reduction maps of various textures. "Sampling" is the time required to sample the reflectance function for each base-level texel. "NNLS" is the time it took to run the NNLS. The time it took to fill in each texel's weights and normal distributions are found in "Filling." The total time to run the algorithm is in "Total." Note that with the exception of the bump-mapped texture, all textures had a function to return the shader normal.

The speed overhead of running the procedural reduction map algorithm is demonstrated in Fig. 9. To calculate this, the procedural shader returns only white with no computations, effectively eliminating the cost of texture calls from the timings. This leaves two costs: the ray/object intersection tests and the cost of running the procedural reduction map algorithm per 1,000,000 pixels (calculated by subtracting the timing for 1-Sample from the timing for the PRM). To better display the cost of running the procedural reduction map algorithm, we use two models with different numbers of triangles, as well as for each model using two different numbers (k) of basis reflectance functions. The normal distribution map version was not included in this since it is a blend of the procedural reduction map and regular supersampling methods.

The animated scene with many procedural shaders (shown in Fig. 1 and in the accompanying video) has 840 frames and took 10.5 hours to render at 16 samples per pixel, whereas it took only 3.0 hours to render using procedural reduction maps (using 1 sample per pixel). Note that the 16 samples per pixel version still aliased at this rate. No minification aliasing can be seen in the PRM version of this sequence. Furthermore, over the 840 frames, the 16 samples per pixel version made more than 280 million texture calls, whereas the procedural reduction map only made 60 million.

7 Results for GPU Rendering

We made use of an Nvidia Quadro NVS 280 PCI-Express graphics card with 64 Mbytes of memory to test the use of procedural reduction maps on the GPU. We used the shader language "Sh" to implement our test shader [40].

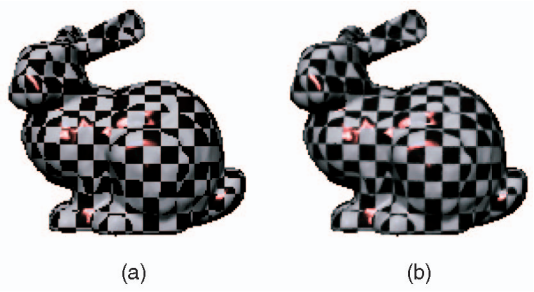

Fig. 11. GPU rendering. (a) An original shader (18 fps). (b) Using a procedural reduction map (16 fps).

Even though this rendering was performed on the GPU, our reflectance sampling and analysis was done in the CPU. We made this choice because of the large infrastructure that would have been required to execute the reflectance sampling on the GPU and to read these samples back into the CPU.

Fig. 11 shows a comparison of a procedural 3D checkerboard that was rendered from the original shader (Fig. 11a) and that was rendered using a procedural reduction map (Fig. 11b). Note that the dark and light squares of the checkerboard use two different specular exponents and colors. Even in the still image, the antialiased boundaries between the dark and light squares are evident. In the accompanying video, strong rendering artifacts due to the checkerboard are apparent in the original shader results, and these are gone in the procedural reduction map results. Because both video sequences were created using one sample per pixel, silhouette aliasing is still evident in both sequences.

8 Conclusion and Future Work

The procedural reduction map provides an automatic method of antialiasing procedural shaders. In addition to handling simple color variation, this approach also antialiases reflectance variations due to specular highlights. We believe this to be a general tool that will relieve the author of many procedural textures from the difficult task of writing code to antialias textures.

There are several opportunities for future work on this problem. One topic is how to treat anisotropic reflectance functions. Handling the extra dimension in an inexpensive manner could prove difficult. Another direction is to investigate antialising of time-varying textures. The obvious issues that this brings up are the high cost of precomputation and the potential for high-memory consumption. Another open issue is speeding up the treatment of shiny and bumpy surfaces during rendering, since this is commonplace among procedural shaders. Along similar lines, it would be good to allow specular highlights that are not centered around the reflection vector. Currently, reflectance functions that have specular highlights that are not near the reflection vector would not be distinguishable. The obvious solution is to more heavily sample the reflectance function but at a high cost in compute time. Finally, we want to find a fast and accurate method of integrating the reflectance function for those cases when the variance of the normal distribution is high.

Acknowledgments

The authors thank Eugene Zhang for providing models with excellent texture atlases. They thank Mitch Parry for his assistance with matrix factorization algorithms. This research was supported in part by US National Science Foundation (NSF) Grants CCF-0204355 and CCF-0625264.

References

[1] S. Upstill, *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley Longman, 1989.
[2] M. Olano, J.C. Hart, W. Heidrich, and M. McCool, *Real-Time Shading.* A.K. Peters, 2002.
[3] L. Williams, "Pyramidal Parametrics," *Computer Graphics,* vol. 17, no. 3, 1983.
[4] A.A. Apodaca and L. Gritz, *Advanced RenderMan: Creating CGI for Motion Pictures.* Morgan Kaufmann, 2000.
[5] N. Carr and J.C. Hart, "Mesh Atlases for Real-Time Procedural Solid Texturing," *ACM Trans. Graphics,* pp. 106-131, 2002.
[6] R.L. Cook, "Shade Trees," *Proc. ACM SIGGRAPH '84,* pp. 223-231, 1984.
[7] D. Peachey, "Solid Texturing of Complex Surfaces," *Proc. ACM SIGGRAPH '85,* pp. 279-286, 1985.
[8] K. Perlin, "An Image Synthesizer," *Proc. ACM SIGGRAPH '85,* pp. 287-296, 1985.
[9] D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling: A Procedural Approach,* third ed. Morgan Kaufmann, 2003.
[10] A. Norton and A.P. Rockwood, "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space," *Computer Graphics,* vol. 16, no. 3, 1982.
[11] P.S. Heckbert, "Filtering by Repeated Integration," *Proc. ACM SIGGRAPH '86,* pp. 315-321, 1986.
[12] W. Heidrich, P. Slusallek, and H.-P. Seidel, "Sampling Procedural Shaders Using Affine Arithmetic," *ACM Trans. Graphics,* vol. 17, no. 3, pp. 158-176, 1998.
[13] D. Goldman, "Fake Fur Rendering," *Proc. ACM SIGGRAPH '97,* pp. 127-134, 1997.
[14] J.C. Hart, N. Carr, M. Kameya, S.A. Tibbitts, and T.J. Coleman, "Antialiased Parameterized Solid Texturing Simplified for Consumer-Level Hardware Implementation," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (HWWS '99),* pp. 45-53, 1999.
[15] M. Olano, B. Kuehne, and M. Simmons, "Automatic Shader Level of Detail," *Graphics Hardware,* pp. 7-14, 2003.
[16] F. Pellacini, "User-Configurable Automatic Shader Simplification," *Proc. ACM SIGGRAPH '05/ACM Trans. Graphics,* pp. 445-452, 2005.
[17] J. Lawrence, A. Ben-Artzi, C. DeCoro, W. Matusik, H. Pfister, R. Ramamoorthi, and S. Rusinkiewicz, "Inverse Shade Trees for Non-Parametric Material Representation and Editing," *Proc. ACM SIGGRAPH '06/ACM Trans. Graphics,* vol. 25, no. 3, July 2006.
[18] J. Kautz and H.-P. Seidel, "Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (HWWS '00),* pp. 51-58, 2000.
[19] D.K. McAllister, A. Lastra, and W. Heidrich, "Efficient Rendering of Spatial Bi-Directional Reflectance Distribution Functions," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (HWWS '02),* pp. 79-88, 2002.
[20] J. Amanatides, "Ray Tracing with Cones," *Proc. ACM SIGGRAPH '84,* pp. 129-135, 1984.
[21] B.G. Becker and N.L. Max, "Smooth Transitions Between Bump Rendering Algorithms," *Proc. ACM SIGGRAPH '93,* pp. 183-190, 1993.
[22] T. Malzbender, D. Gelb, and H. Wolters, "Polynomial Texture Maps," *Proc. ACM SIGGRAPH '01,* pp. 519-528, 2001.
[23] A. Fournier, "Filtering Normal Maps and Creating Multiple Surfaces," *Proc. Graphics Interface Workshop Local Illumination,* pp. 45-54, 1992.

[24] M. Olano and M. North, "Normal Distribution Mapping," Univ. of North Carolina Computer Science Technical Report 97-041, 1997.
[25] P. Tan, S. Lin, L. Quan, B. Guo, and H.-Y. Shum, "Multiresolution Reflectance Filtering," *Proc. Eurographics Symp. Rendering,* pp. 111-116, 2005.
[26] K. Perlin, "Improving Noise," *Proc. ACM SIGGRAPH '02,* pp. 681-682, 2002.
[27] R.L. Cook and T. DeRose, "Wavelet Noise," *Proc. ACM SIGGRAPH '05,* pp. 803-811, 2005.
[28] P.S. Heckbert, "Survey of Texture Mapping," *IEEE Computer Graphics and Applications,* vol. 6, pp. 56-67, 1986.
[29] F.C. Crow, "Summed-Area Tables for Texture Mapping," *Proc. ACM SIGGRAPH '84,* vol. 18, pp. 207-212, 1984.
[30] A. Schilling, G. Knittel, and W. Strasser, "Texram: A Smart Memory for Texturing," *IEEE Computer Graphics and Applications,* vol. 16, no. 3, pp. 32-41, 1996.
[31] H. Igehy, "Tracing Ray Differentials," *Proc. ACM SIGGRAPH '99,* pp. 179-186, 1999.
[32] P. Sander, J. Snyder, S. Gortler, and H. Hoppe, "Texture Mapping Progressive Meshes," *Proc. ACM SIGGRAPH '01,* pp. 409-416, 2001.
[33] E. Zhang, K. Mischaikow, and G. Turk, "Feature-Based Surface Parameterization and Texture Mapping," *ACM Trans. Graphics,* pp. 1-27, 2005.
[34] C. Lawson and R. Hanson, *Solving Least Square Problems.* Prentice Hall, 1974.
[35] P. Schröder and W. Sweldens, "Spherical Wavelets: Efficiently Representing Functions on the Sphere," *Proc. ACM SIGGRAPH '95,* pp. 161-172, 1995.
[36] M. McCool, J. Ang, and A. Ahmad, "Homomorphic Factorization of BRDF's for High Performance Rendering," *Proc. ACM SIGGRAPH '01,* pp. 185-194, 2001.
[37] J. Lawrence, S. Rusinkiewicz, and R. Ramamoorthi, "Efficient BRDF Importance Sampling Using a Factored Representation," *Proc. ACM SIGGRAPH '04,* pp. 495-505, 2004.
[38] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen, "The Lumigraph," *Proc. ACM SIGGRAPH '96,* pp. 43-54, 1996.
[39] T.E. Zickler, P.N. Belhumeur, and D.J. Kriegman, "Helmholtz Stereopsis: Exploiting Reciprocity for Surface Reconstruction," *Int'l J. Computer Vision,* vol. 49, 2002.
[40] M. McCool and S. DuToit, *Metaprogramming GPUs with Sh.* A.K. Peters, 2004.
[41] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, "A General Method for Preserving Attribute Values on Simplified Meshes," *Proc. IEEE Visualization,* pp. 59-66, 1998.
[42] M. Rioux, M. Soucy, and G. Godin, "A Texture-Mapping Approach for the Compression of Colored 3D Triangulations," *The Visual Computer,* vol. 12, no. 10, pp. 503-514, 1996.
[43] A.P. Witkin, "Recovering Surface Shape and Orientation from Texture," *Artificial Intelligence,* vol. 17, pp. 17-45, 1981.
[44] S. Worley, "A Cellular Texture Basis Function," *Proc. ACM SIGGRAPH '96,* pp. 291-294, 1996.

R. Brooks Van Horn III received the PhD degree in computer science from the Georgia Institute of Technology in 2007, where he was a member of the College of Computing and the Graphics, Visualization, and Usability Center. He served twice as an intern at Pixar Animation Studios. His research interests include photo-realistic rendering and animation.

Greg Turk received the PhD degree in computer science from the University of North Carolina, Chapel Hill, in 1992. He is currently an associate professor at the Georgia Institute of Technology, where he is a member of the College of Computing and the Graphics, Visualization, and Usability Center. His research interests include computer graphics, scientific visualization, and computer vision. He is a member of the IEEE.