Real-Time Bump Map Synthesis

Jan Kautz* Wolfgang Heidrich† Hans-Peter Seidel*

Max-Planck-Institut für Informatik* University of British Columbia†

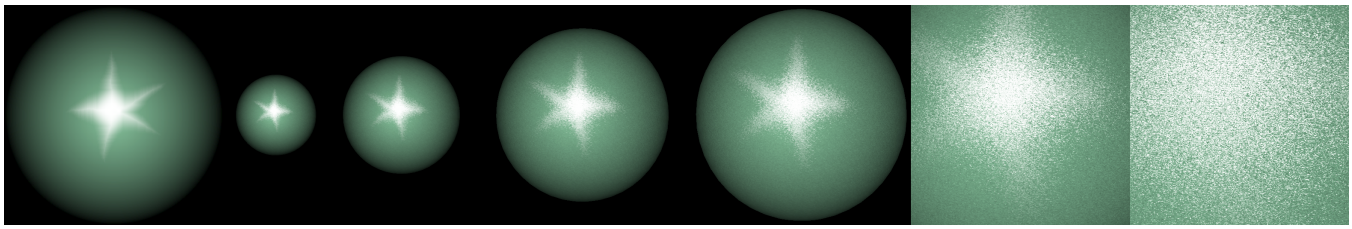

Figure 1: *Bump mapped sphere at different levels of detail consistently generated and shaded with the normal distribution shown on the left.*

Abstract

In this paper we present a method that automatically synthesizes bump maps at arbitrary levels of detail in real-time. The only input data we require is a normal density function; the bump map is generated according to that function. It is also used to shade the generated bump map.

The technique allows to infinitely zoom into the surface, because more (consistent) detail can be created on the fly. The shading of such a surface is consistent when displayed at different distances to the viewer (assuming that the surface structure is self-similar).

The bump map generation and the shading algorithm can also be used separately.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and frame buffer operations; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing and Texture

1 Introduction

Bump mapping was originally introduced by Blinn [3] in 1978. He showed how wrinkled surfaces can be simulated by only perturbing the normal vector, without changing the underlying surface itself. The perturbed normal is then used for the lighting calculations instead of the original normal.

Current graphics hardware usually supports bump mapping by providing per-pixel operations such as dot-products accessible through either OpenGL extensions [18] or DirectX 8 [16]. These per-pixel operations are very flexible and allow not only bump maps with diffuse and specular reflections using a Lambertian reflection model or resp. the Blinn-Phong model [4, 12, 15], but even using more complex reflection models [14].

Bump maps are usually created by artists to create the illusion of certain surfaces or surface structures. In many cases these bump maps just contain more or less random bumps to better simulate rough materials at close-ups. The perceived roughness of such surfaces should be the same when displayed at different distances to the viewer. But shading of these bump mapped surfaces is often inconsistent across different levels of detail, because the shading model at the coarsest level of detail does not correspond to the bump map at finer level of details. Or put the other way round, the bump map is not consistent with the chosen shading model.

We present a method that automatically generates bump maps in real-time given only a shading model. It performs consistent shading under the assumption that the wrinkled surface is fractal and that the reflection model is based on microfacets (i.e. many tiny specular surface patches). This also allows to infinitely zoom into the surface, because more (consistent) detail can be created on the fly.

The basic idea of our method is simple. Since we presume that the shading model is based on microfacets, we can create a bump map that is consistent with the original shading model by distributing bumps according to the *normal density function* (NDF), which describes the probability distribution of different microfacet orientations. When the viewer gets closer to the bump map, we can create more wrinkles at higher frequencies while maintaining the normal distribution of the shading model; see Figure 1. The creation of bumps is governed by a noise function. The shading is always done with the original shading model, which is (almost) correct, because the generated detail is self-similar (fractal); no matter how close one zooms in, the microfacets always have the same distribution.

Consequently, we are synthesizing finer detail from a very coarse description. It is worth noting that this is different from most existing texture synthesis methods, where data is synthesized at same level of detail. Our main contributions are

- a method that interactively synthesizes bump maps at arbitrary levels of detail from a normal density function, and
- a rendering algorithm that performs shading of the generated bump maps with a reflectance model based on the same normal density function.

These two parts can actually be used independently.

*{kautz,hpseidel}@mpi-sb.mpg.de, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany.

†heidrich@cs.ubc.ca, Dept. of Computer Science, 3645 - 2424 Main Mall, Vancouver, BC, V6T 1Z4, Canada

In the following section we will briefly review some related work. In Section 3 we will first give an overview of our method and then detail the bump map generation in Section 4 and the rendering algorithm in Section 5. After presenting some results in Section 6 we show possible extensions (Section 7) and finally conclude (Section 8).

2 Related Work

While bump mapping has been around for a while [3], implementations using graphics hardware have been proposed more recently. A technique called embossing [5] uses a multipass method that works on traditional graphics hardware. However, dot-product bump mapping is preferred nowadays because it produces better results and is more flexible, although it needs hardware support for advanced per-pixel operations. It directly stores the normals of the surface in texture maps [12, 30] and can be used to render diffuse and specular reflections from small surface bumps.

Special hardware for bump mapping has also been proposed or even implemented [7, 17, 21]. Olano and Lastra [19] used a more general approach, they have built graphics hardware that can run small shading programs and hence is capable of doing bump mapping.

Many reflectance models have been introduced in computer graphics the past years. We will only briefly mention those that are closely related to our work. Blinn [4] proposed a very simple shading model based on microfacets, which is often used in the context of bump mapping. It is a modification of the commonly used Phong model [23] to make it visually more satisfying. The Cook-Torrance model [6] is also based on microfacets, but adds a shadowing and Fresnel term to make it more realistic. Ward [28] introduced an anisotropic BRDF model that is based on an anisotropic Gaussian microfacet distribution. Ashikmin et al. [1] have recently introduced a way of generating reflection models from arbitrary normal distributions.

Recently new techniques have been developed to incorporate more complex BRDF models into real-time rendering. For example, the Banks model [2] and the Cook-Torrance model [6] were used by Heidrich and Seidel [12]. More general reflectance models were used for hardware accelerated shading by Kautz and McCool [13]. These methods cannot be used for bump mapping, because they assume smoothly varying surface normals. Kautz and Seidel introduced a method [14] that combines bump mapping with more complex analytic BRDFs, but did not address the issue of mipmapping or filtering in general.

An effective way to do mipmapping of bump maps (in software) was introduced by Fournier [8]. A similar technique was proposed by Olano and North [20]. Our method tries to solve a different problem, but performs consistent shading for different mipmap levels assuming the surface structure is fractal.

There is also a host of work on texture synthesis, both in the computer graphics and in the computer vision literature. Common approaches include feature matching (e.g. [10]), Markov random fields (e.g. [29]), and physical simulations such as reaction-diffusion models [31]. These algorithms synthesize global patterns that are much more complex than the simple fractal patterns we are considering. However, these algorithms are far from realtime synthesis and only partly amenable to level-of-detail representations.

3 Overview

As our method is based on a fractal microfacet model, we first provide a summary of such models before we give an overview of the proposed algorithm.

3.1 Microfacet-Based Reflectance Models

Microfacet-based reflectance models assume that a surface is made of many small, flat, and perfectly specular patches (Fresnel reflectors), so-called *microfacets* (see [6] for a more detailed discussion). These microfacets only reflect light in the specular direction with respect to its own normal $\hat{n}_m$. The overall appearance of the surface is governed by the distribution of the orientation of microfacets, given as a probability density function $p(\hat{n}_m)$, also called the normal density function (NDF); see Figure 2.

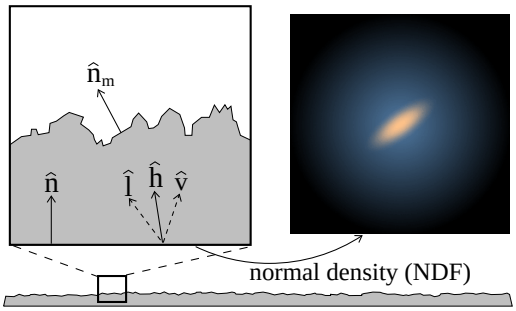

Figure 2: *Microfacet-based surfaces. $\hat{n}_m$ is the normal of a microfacet, $\hat{n}$ is the normal of the surface, and $\hat{v}$ and $\hat{l}$ are exemplary local viewing and light directions. On the right you can see a visualization of a normal density function $p(\hat{n}_m)$.*

This NDF can be used directly for shading purposes, given that we are far enough away so that we cannot discern individual microfacets. The normalized half-way vector $\hat{h}$ between the eye vector $\hat{v}$ and the light vector $\hat{l}$ can be used to look up the fraction of microfacets that will reflect light towards the eye for the given light and eye vector. This value $p(\hat{h})$ can be directly used for shading, as it is done for example by the Blinn-Phong model [4] or in [8, 20]. More accurate models [27, 4, 6, 8, 1] include a self-shadowing/masking term and a Fresnel term. In our work we currently do not include these terms, simply for efficiency reasons. But inclusion of these terms is possible using a combination of [14] and [1].

3.2 Fractal Surfaces

We assume the surface to be made of fractal microfacets. A fractal microfacet surface can be described as a surface consisting of microfacets that are not perfectly specular but themselves consist of microfacets, which happen to be distributed according to the same density function, and so on. This implies that when we get close enough to the surface to see individual microfacets, we have to perform shading of the microfacets with the reflectance model based on the NDF. When we get even closer we have to generate more microfacets distributed according to the NDF.

3.3 Algorithm Overview

The problem we are trying to solve is the following. Instead of requiring an artist to draw a bump map, which would only be valid for one level of detail, we want to synthesize a bump map at arbitrary levels of detail from a given reflectance model. The reflectance model must be based on microfacets and the probability density function $p(\hat{n}_m)$ of the microfacet distribution must be given.

This density function tells us what percentage of microfacets is oriented in which way. So synthesizing a bump map requires the creation of a normal map that is consistent with the density function. Then this bump map, represented as a normal map, has to be

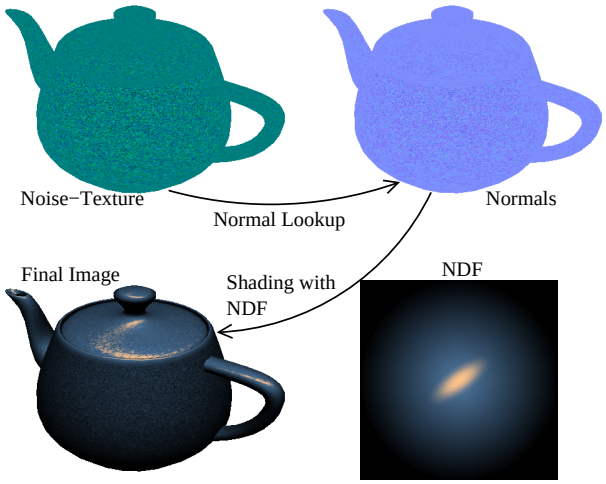

Figure 3: *Conceptual overview of our method. First, the object is rendered with distance-dependent fractal noise (2D Gaussian distributed random numbers). Then a lookup is performed that returns local surface normals defining a bump map; the lookup table is generated in such a way that the normals are distributed according to the NDF. Finally the NDF is used to shade the bump map.*

shaded. Since we assume the surface to be fractal, shading is done with a microfacet reflection model based on the same NDF.

We are not explicitly creating the normal map, we rather generate it on-the-fly. The generation of a normal map according to a probability density function can be seen as the generation of a random variable with a given density. This boils down to precomputing a lookup table that takes a two-dimensional, $[0,1]$-distributed random vector as its input and outputs a normal (more detail about this can be found in the next section). The lookup table is generated in such a way that normals are distributed according to the NDF.

Rendering is straightforward then. A distance-dependent noise texture is generated for the object, and the values from the noise texture and the precomputed table are used to look up normals defining a bump map. Then the NDF-based reflection model is used for shading this generated bump map.

Different levels of detail depending on the distance to a surface are generated as follows. When the object is seen from far away, no individual bumps should be visible, i.e. the 2D noise texture should contain a vector (depends on the representation of the NDF) that looks up the normal $(0,0,1)$. When getting closer to the surface more detail will become visible, which can be e.g. achieved by smoothly blending in Perlin noise [22]. The closer you get the more octaves of Perlin noise should be added, resulting in more detail when needed.

In Figure 3 the individual steps are visualized.

4 Bump Map Generation

As explained in the previous section, we want to build a lookup table that maps 2D random vectors to normals according to the normal density function. This can be seen as the generation of a random variable with a given density. This is a well-known problem [24], and is generally solved the following way.

Given a two dimensional probability density function (PDF) $f(x,y)$ and uniformly $[0,1]$-distributed random numbers (r_1, r_2), we first compute the marginal density function:

$$m(x) := \int_{-\infty}^{\infty} f(x,y)dy,$$

and then its cumulative distribution function (CDF):

$$M(x) := \int_{-\infty}^{x} m(x')dx'.$$

Given a uniformly distributed random number r_1, we can compute a new random number x_n that is distributed according to $f(x,y)$ in the following manner:

$$x_n := M^{-1}(r_1), \quad M^{-1} \text{ being the inverse CDF.}$$

Now it is necessary to compute the following conditional PDF for a given x_n:

$$c(y|x_n) := \frac{f(x_n,y)}{m(x_n)},$$

and its cumulative distribution function:

$$C(y|x_n) := \int_{-\infty}^{y} c(y'|x_n)dy',$$

which can then be used to compute the other new random number:

$$y_n := C^{-1}(r_2|x_n), \quad C^{-1} \text{ being the inverse CDF.}$$

The random numbers (x_n, y_n) are also in $[0,1]$, but distributed according to the PDF $f(x,y)$.

This works if the initial random numbers (r_1, r_2) are uniformly $[0,1]$-distributed. If the (independent) initial random numbers r_1 and r_2 are distributed according to some density function $g(x)$, we first have to map them to uniformly distributed random numbers and then apply the computation from above. The mapping is done the following way:

$$r_1^* := G(r_1), \quad r_2^* := G(r_2),$$

where (r_1^*, r_2^*) are the new uniformly distributed random variables, $G(x)$ is the cumulative distribution function of $g(x)$. Combining both mappings gives

$$\begin{aligned} x_n &:= M^{-1}(G(r_1)), \\ y_n &:= C^{-1}(G(r_2)|x_n). \end{aligned}$$

We use several octaves of Perlin noise to synthesize a turbulence function (i.e. fractal noise). Since both Perlin noise and the resulting turbulence function are known to produce random numbers (r_1, r_2) with a Gaussian distribution [25], we have to do the additional mapping described above.

The above computations can be used directly to create a normal map according to a normal density function from our Gaussian distributed random numbers. The NDF is two-dimensional depending on the x and y-coordinate of $\hat{n}_m$, and hence the same technique can be applied. Since we store our normal distribution functions in a 2D hemispherical map (directly using the x- and y-component of the normals, see right side of Figure 2), all the above computations are done numerically. The actual distribution $g(x)$ of the Perlin turbulence is also computed numerically. We assume that the Gaussian distribution of the Perlin turbulence remains constant independent of the number of octaves, which is a valid approximation. The resulting random numbers (x_n, y_n) are expanded to a normal describing the direction of a microfacet:

$$\hat{n}_m = \left(2x_n - 1, 2y_n - 1, \sqrt{1 - (2x_n - 1)^2 - (2y_n - 1)^2}\right)^T.$$

We generate a full lookup table $L(r_1, r_2)$ with 256×256 entries for mapping (r_1, r_2) to $\hat{n}_m$. This is done once for every normal density function in a preprocessing step. In addition we are deterministically generating a tangent frame for every normal $\hat{n}_m$, which we also store in the lookup table. A complete tangent frame is needed for anisotropic NDFs, as seen for example in Figure 1, otherwise the normal is sufficient. The lookup table is then used during rendering.

5 Rendering

In this section we will detail how the rendering is done. We will first show how this technique can be used for software rendering, and then how the same method can be implemented on current graphics hardware (using an NVIDIA GeForce 3).

5.1 Software

A software renderer (e.g. raytracer) can easily implement this technique. The rendering algorithm for a single ray works as follows. When the ray hits the object, we compute the distance d from the eye to the intersection point. If the distance is above some (user-defined) threshold, then the object is not close enough to discern its surface structure, hence we just shade the surface with the NDF (i.e. no bump mapping at all). Alternatively, we could use a user-provided bump map with an arbitrary normal distribution as the top level for macroscopic features.

If the distance is below some threshold, bumps start appearing. We then evaluate a noise function for that surface — a fractal noise function such as Perlin turbulence should be used. We adjust the number of used octaves depending on the distance d, the closer the more octaves we use. Whenever a new octave is added, it should be blended in smoothly to avoid popping artifacts. The noise function has to be evaluated twice (with different settings) to get two different random numbers (r_1, r_2). These are used to look up a (local) microfacet tangent frame (normal $\hat{n}_m$, tangent $\hat{t}_m$, and binormal $\hat{b}_m$) using the table $L(r_1, r_2)$.

Then we compute the halfway vector $\hat{h}$ between the viewing and light vector and express it in the local surface tangent frame, which must be provided by the object's model. Now both the microfacet tangent frame and the half-way vector $\hat{h}$ are in local coordinates relative to the local surface frame. As stated before, we want to shade the microfacets (defined by the tangent frames $\{\hat{t}_m, \hat{b}_m, \hat{n}_m\}$) with the NDF and not the original surface. Since the NDF is indexed with the half-way vector in local coordinates, we have to project the half-way vector $\hat{h}$ into the microfacet's tangent frame resulting in $\hat{h}_m$. Now we can lookup the NDF $p(\hat{h}_m)$ and use the stored value for shading.

See the Section 6 for results using software rendering.

5.2 Hardware

In this section we will describe how our bump map synthesis method can be implemented on current graphics hardware.

5.2.1 Dependent Texture Lookup

In order to implement our technique we need dependent texturing, a feature now supported on modern graphics hardware. Dependent texturing allows the entries of one texture map to be used as texture coordinates for a lookup into a second texture map.

We implemented our technique on an NVIDIA GeForce 3, the only currently available graphics card with a fairly flexible dependent texture lookup[1]. We will briefly outline how this works on GeForce 3 cards.

The dependent texture lookup is embedded in the texture shaders, a new stage in the pipeline which takes place before multitexturing, i.e. also before NVIDIA's register combiners. The texture shaders are divided into four stages. Every stage takes a texture map from the corresponding texture unit as its input, as well as the texture coordinate set from that texture unit. Other potential inputs are the results from one or two previous stages. Every stage runs one of about 20 canned programs. The programs include normal texturing, dependent texture lookup from the green and blue channel of a previous stage, computation of dot-products, dependent lookup into a 2D texture using results of two dot-products, and many more.

Please note that since the texture shaders only support four stages, we can perform shading with isotropic NDFs only. For the hardware rendering we store the isotropic NDF $p_i(\hat{n}_m \cdot \hat{h}_m)$ in a 1D texture. An additional stage would allow us to implement anisotropic distributions as well.

5.2.2 Rendering Algorithm

The algorithm consists of multiple parts: noise generation, the normal lookup using $L(r_1, r_2)$, and shading with the NDF. We explain all three parts here.

Ideally, the graphics hardware would support a procedural noise function, for example using an implementation similar to the one proposed by Hart et al. [9]. Since this is not (yet) the case, we have two possibilities to texture an object with distance-dependent noise. The first possibility is to implement Perlin turbulence with a multipass algorithm [5], which is expensive. We chose to use a less expensive way. We create a mipmapped texture containing Perlin turbulence that is applied to the object. The finest mipmap level contains the most octaves, every coarser mipmap level contains one octave less. This is done to fade out the bumpy appearance of the surface when it is viewed from far away.

The single-pass rendering algorithm uses NVIDIA's texture shader extension, which provides the dependent texture lookup. It works as follows. We load the noise texture into texture unit 0 (the green and blue component contain r_1 and r_2), the lookup texture $L(r_1, r_2)$ into texture unit 1, and the NDF into texture unit 3. In contrast to the software rendering method, the lookup table only contains normals $\hat{n}_m$, which is because the texture shaders are limited to four stages.

We then set up the texture shader as follows. Texture shader stage 0 performs standard texture mapping with the noise texture. Texture shader stage 1 takes the green and blue component of the noise texture and looks up $L(r_1, r_2)$ resulting in a microfacet normal $\hat{n}_m$. The texture coordinates for texture unit 2 are set to the halfway vector $\hat{h}$ (with respect to the surface tangent frame). Texture shader stage 2 is set to compute the dot-product between the texture coordinates, i.e. $\hat{h}$, and the result from stage 1, i.e. the normal $\hat{n}_m$. The last texture shader stage then uses the result of this dot-product to look up a 1D texture map containing the isotropic NDF (actually the texture shaders require a lookup into a 2D texture, we simply set the second coordinate to zero). The result from this lookup is $p(\hat{n}_m \cdot \hat{h}_m)$, which is then directly used to texture the surface.

If the texture shaders supported one more stage, it would be possible to implement shading with anisotropic NDFs. Other vendors are expected to incorporate a similar stage (required by DirectX 8 [16]), and hopefully it will be more flexible in the future, so that anisotropic NDF shading can be implemented as well.

[1] SGI Octanes also support dependent texturing (called Pixel-Textures) but only via a framebuffer copy.

Figure 5: *These images were generated at 30Hz on an AMD Athlon 1GHz and an NVIDIA GeForce 3.*

Anisotropic shading would also be possible if we did the lookup $L(r_1, r_2)$ in advance and directly stored the normals in texture maps freeing a stage in the texture shaders. We decided not to do so since bilinear filtering of normals often leads to artifacts.

6 Results

We now would like to show a few results generated with this technique.

In Figure 4 you can see a teapot rendered in software using our technique with an anisotropic NDF. On the left you can see the NDF, the three middle images show renderings at different distances. The image on the right shows what happens if the normals are distributed uniformly and not according to the NDF.

In Figure 5 you can see two renderings done with an isotropic NDF using NVIDIA's GeForce 3. The teapot model consists of about 17000 triangles. Since the generation of the bump map and the shading can be done in a single-pass, it was possible to render about 30 frames per second. The resulting quality is very high. In particular, it is almost aliasing free. The main reason for this is that the indices (r_1, r_2) are filtered and the filtered indices are used for the lookup to get a normal. This avoids problems of unnormalized and degenerated normals that may occur with normal mapping. Unfortunately, it is somtimes possible to see some ringing in the shading when the object is moved around. We are not entirely sure where it comes from, but it is probably due to quantization errors (e.g. the indices in the noise-texture are limited to 8 bits by the hardware). It does not occur in the software renderings.

All our renderings were done with the NDF only. As can be seen in the images, a diffuse component (although not a Lambertian diffuse component) can be incorporated directly into the NDF. Alternatively, it is also possible to add a separate diffuse bump mapping term that is computed conventionally, which has the advantage that a diffuse texture can be modulated with it.

We noticed that some care should be taken when generating the Perlin noise. We never used low-frequency noise generated by only a few octaves, because the resulting bumps were much too large.

7 Extensions

This technique can be easily extended in different ways.

First of all, the bump map synthesis and the bump map NDF shading can be used separately. For example, the generated bump map can be used with some other shading technique, e.g. [14]. If the bump map NDF shading is used separately, it can be implemented in hardware for anisotropic NDFs and not only isotropic NDFs, since two additional texture stages are free. However, using an arbitrary hand-crafted bump map will result in inconsistencies between the bump map and the refection model.

Often it might be desirable to add more detail to a surface that already uses a bump map. It is easily possible to do this with our method. The original bump map just needs to be converted to a texture containing indices for the lookup table $L(r_1, r_2)$ so that the original normals will be looked up. This can be done using the following computation: $r_1 = G^{-1}(M(n_x))$ and $r_2 = G^{-1}(C(n_y|n_x))$, where $\hat{n} = (n_x, n_y, n_z)$ is the normal from the bump map, G is the cumulative distribution of the noise density, M, and C are the marginal, resp. conditional distribution functions; see Section 4. When zooming closer to the surface, noise can be added to this index texture. This combined index texture is then used for the lookup. On a GeForce 3 it is not possible to easily implement this extension, since the texture shader stage does not allow to add two textures before they are used for a dependent lookup, future hardware will hopefully allow more flexibility in this stage.

Another extension is to generate an NDF for an existing bump map and then use the NDF for shading of the bump map. Again under the assumption of a fractal surface, this will produce consistent shading across different levels of detail. Of course this is still not the same as correctly mipmapping a bump map as proposed by Fournier [8].

Since multitexturing is not used by our algorithm, one can easily add a Fresnel term (at least an approximation to it), and potentially also a shadowing/masking term depending on how complex it is.

8 Conclusion and Discussion

We have presented a method to automatically generate detail in the form of bump maps given only a shading model, i.e. a normal density function to be more precise. Our technique generates more detail when the user gets closer to a surface and more of the surface structure becomes visible. The algorithm assumes an underlying fractal surface structure. Under this assumption the shading is consistent across different levels of detail.

The technique can either be implemented in software or using modern graphics hardware working in real-time. The hardware implementation is restricted in two ways, the shading works only with isotropic NDFs and the necessary noise generation is not available in hardware, so it was simulated by precomputing noise and storing it in mipmaps. The disadvantage of doing so is that either you have to compute a very large texture or repeat the texture over the surface in order to be able to zoom in closely; otherwise individual microfacets can become visible breaking the assumption of a fractal surface. Hopefully procedural noise will be available in future hardware, as it can be used in many different ways to synthesize detail and hence saving bandwidth from the host to the graphics card.

Recently techniques have been proposed to cast shadows in bump maps [11, 26] and to compute scattering in bump maps [11]. We have not considered shadows or scattering.

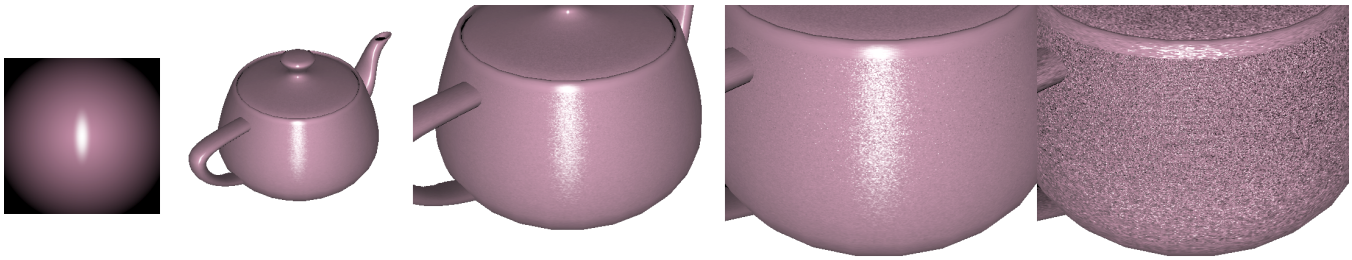

Figure 4: *Teapot with automatically generated bump map using the NDF shown on the left. On the far right the teapot was rendered with uniformly distributed normals, i.e. not according to the NDF.*

9 Acknowledgements

We would like to thank the anonymous reviewers for their detailed and helpful comments. Furthermore, we would like to thank NVIDIA for giving us early access to a GeForce 3 card. The first author would like to thank the Imager Lab at UBC for making his stay in Vancouver so pleasant.

References

[1] ASHIKHMIN, M., PREMOZE, S., AND SHIRLEY, P. A Microfacet-based BRDF Generator. In *Proceedings SIGGRAPH* (July 2000), pp. 65–74.

[2] BANKS, D. Illumination in Diverse Codimensions. In *Proceedings SIGGRAPH* (July 1994), pp. 327–334.

[3] BLINN, J. Simulation of Wrinkled Surfaces. In *Proceedings SIGGRAPH* (Aug. 1978), pp. 286–292.

[4] BLINN, J. Models of Light Reflection For Computer Synthesized Pictures. In *Proceedings SIGGRAPH* (July 1977), pp. 192–198.

[5] BLYTHE, D., GRANTHAM, B., MCREYNOLDS, T., AND NELSON, S. Advanced Graphics Programming Techniques Using OpenGL. In *SIGGRAPH '00 Course Notes* (July 2000).

[6] COOK, R., AND TORRANCE, K. A Reflectance Model for Computer Graphics. In *Proceedings SIGGRAPH* (Aug. 1981), pp. 307–316.

[7] ERNST, I., RÜSSELER, H., SCHULZ, H., AND WITTIG, O. Gouraud Bump Mapping. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware* (1998), pp. 47–54.

[8] FOURNIER, A. Normal Distribution Functions and Multiple Surfaces. In *Graphics Interface '92 Workshop on Local Illumination* (May 1992), pp. 45–52.

[9] HART, J., CARR, N., KAMEYA, M., TIBBITTS, A., AND COLEMAN, T. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware 1999* (Aug. 1999), pp. 45–54.

[10] HEEGER, D., AND BERGEN, J. Pyramid-based texture analysis/synthesis. In *Proceedings SIGGRAPH* (Aug. 1995), pp. 229–238.

[11] HEIDRICH, W., DAUBERT, K., KAUTZ, J., AND SEIDEL, H.-P. Illuminating Micro Geometry Based on Precomputed Visibility. In *Proceedings SIGGRAPH* (July 2000), pp. 455–464.

[12] HEIDRICH, W., AND SEIDEL, H. Realistic, Hardware-accelerated Shading and Lighting. In *Proceedings SIGGRAPH* (Aug. 1999), pp. 171–178.

[13] KAUTZ, J., AND MCCOOL, M. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *Tenth Eurographics Workshop on Rendering* (June 1999), pp. 281–292.

[14] KAUTZ, J., AND SEIDEL, H.-P. Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000* (August 2000), pp. 51–58.

[15] KILGARD, M. *A Practical and Robust Bump-mapping Technique for Today's GPUs*. NVIDIA Corporation, April 2000. Available from http://www.nvidia.com.

[16] MICROSOFT CORPORATION. *DirectX 8.0 SDK*, Nov. 2000. Available from http://www.microsoft.com/directx.

[17] MILLER, G., HALSTEAD, M., AND CLIFTON, M. On-the-fly Texture Computation for Real-Time Surface Shading. *IEEE Computer Graphics & Applications 18*, 2 (Mar.–Apr. 1998), 44–58.

[18] NVIDIA CORPORATION. *NVIDIA OpenGL Extension Specifications*, Mar. 2001. Available from http://www.nvidia.com.

[19] OLANO, M., AND LASTRA, A. A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *Proceedings SIGGRAPH* (July 1998), pp. 159–168.

[20] OLANO, M., AND NORTH, M. Normal Distribution Mapping. Tech. Rep. UNC CSTR 97-041, University of North Carolina, Chapel Hill, 1997.

[21] PEERCY, M., AIREY, J., AND CABRAL, B. Efficient Bump Mapping Hardware. In *Proceedings SIGGRAPH* (Aug. 1997), pp. 303–306.

[22] PERLIN, K. An Image Synthesizer. In *Proceedings SIGGRAPH* (July 1985), pp. 287–296.

[23] PHONG, B.-T. Illumination for Computer Generated Pictures. *Communications of the ACM 18*, 6 (June 1975), 311–317.

[24] PITMAN, J. *Probability*. Springer, 1992.

[25] PIXAR. *PRMan Application Note #13: Properties of RenderMan Noise Functions.*

[26] SLOAN, P., AND COHEN, M. Hardware Accelerated Horizon Mapping. In *Eleventh Eurographics Workshop on Rendering* (June 2000), pp. 281–286.

[27] TORRANCE, K., AND SPARROW, E. Theory for Off-Specular Reflection From Roughened Surfaces. *Journal of the Optical Society of America 57*, 9 (Sept. 1967), 1105–1114.

[28] WARD, G. Measuring and modeling anisotropic reflection. In *Proceedings SIGGRAPH* (July 1992), pp. 265–272.

[29] WEI, L., AND LEVOY, M. Fast texture synthesis using tree-structured vector quantization. In *Proceedings SIGGRAPH* (Aug. 2000), pp. 479–488.

[30] WESTERMANN, R., AND ERTL, T. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings SIGGRAPH* (July 1998), pp. 169–178.

[31] WITTKIN, A., AND KASS, M. Reaction-diffusion textures. In *Proceedings SIGGRAPH* (July 1991), pp. 299–308.