Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs

Jan Kautz Hans-Peter Seidel

Max-Planck-Institute for Computer Science*

Abstract

In this paper a technique is presented that combines interactive hardware accelerated bump mapping with shift-variant anisotropic reflectance models. An evolutionary path is shown how some simpler reflectance models can be rendered at interactive rates on current low-end graphics hardware, and how features from future graphics hardware can be exploited for more complex models.

We show how our method can be applied to some well known reflectance models, namely the Banks model, Ward's model, and an anisotropic version of the Blinn-Phong model, but it is not limited to these models.

Furthermore, we take a close look at the necessary capabilities of the graphics hardware, identify problems with current hardware, and discuss possible enhancements.

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

Blinn [2] has shown 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. This technique is generally called bump mapping. A simple lighting model such as the Phong model [21] or the Blinn-Phong model [3] is usually used for the lighting calculations.

Recently, hardware accelerated bump mapping has been introduced, which is capable of rendering diffuse and specular reflections. For the diffuse reflection a lambertian surface is assumed and for the specular reflection the Blinn-Phong model is used, since in both cases only an (exponentiated) dot-product needs to be computed, which can be done with either the OpenGL imaging subset [8, 22] or e.g. with NVIDIA's register combiner extension [11].

These hardware accelerated bump mapping techniques only allow to vary the specular coefficient of the Blinn-Phong model freely (using an additional so-called gloss map, which is simply modulated onto the bump map), whereas the Phong exponent, which controls the sharpness of the highlight, has to remain constant across a polygon.

Moreover, the Blinn-Phong model in its original form can only model a very limited number of materials, hence it is desirable to be able to use other more general reflectance models.

Hardware accelerated techniques, which allow the usage of more complex [8] or almost arbitrary [10] bidirectional reflectance functions (BRDFs) have already been developed. These methods can only handle BRDFs that are shift-invariant across a polygon and the local coordinate frame is only allowed to change smoothly (local coordinate frames are specified per vertex rather than per pixel), i.e. they cannot be used for bump mapping or rapidly changing anisotropy.

The per pixel computation of dot-products is now becoming available even on low-end graphics hardware, such as the NVIDIA GeForce 256. Dependent texture lookup (also known as Pixel Textures [23]), which allows to use one texture map to contain indices, which are then used to index into another texture map, is expected to be readily available by the end of the year.

We propose a technique that evaluates BRDFs on a per-pixel basis using these two capabilities at interactive rates, even if the local surface coordinate frame and the material properties (i.e the parameters of the BRDF) are allowed to vary per pixel. For approximations of some simpler lighting models it is even sufficient to have the possibility to do per-pixel dot-products.

The basic idea we use is simple. A given analytical lighting model is broken up into complex functions that cannot be computed by the graphics hardware. These functions are sampled and stored in texture maps. Each of the functions depends on some parameters which need to be computed from surface and material properties, such as the local coordinate frame, roughness of the surface, and so on. For every pixel of the surface these properties are stored in texture maps. The computation of the parameters of the complex functions depends on the used reflectance model, but requires only simple math, such as dot-products and multiplications, which can be done in hardware, since dependent texturing is used to perform all complex operations. The computed parameters are then used to index into the sampled complex functions using a dependent texture lookup, which effectively evaluates them. Finally the evaluated functions are recombined, depending on how they were broken up, e.g. using multiplication or addition.

This allows us to render surfaces at interactive rates, where the tangents and normals as well as the parameters of the reflectance model can change per pixel, which is equivalent to bump mapping with a shift-variant anisotropic BRDF. For instance, it is possible to render a roughly brushed surface with small bumps or to render a knife with a blade that has only a few scratches.

2 Related Work

Hardware accelerated rendering techniques for bump mapping, which was originally introduced by Blinn [2], have recently been proposed. A technique called embossing [15] uses a multipass method that works on traditional graphics hardware and is capable of rendering diffuse and specular reflections. The bump map needs to be specified as heights in a texture map. Dot-product bump map-

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

ping, which is also called normal mapping, is also used to render diffuse and specular reflections from small surface bumps. It directly stores the normals of the surface in texture maps [8, 28], but needs some more advanced hardware feature, namely the ability to compute dot-products per pixel.

Dedicated hardware support for bump mapping has also been proposed or even implemented [5, 16, 20]. A more general approach was used by Olano and Lastra [19], who have built graphics hardware that can run small shading programs and is therefore also capable of doing bump mapping. Pixel textures have already been shown to be useful in a variety of different applications [9].

Many BRDF models have been proposed over the past years. We will only briefly mention a few that we are using throughout this paper. The widely used Phong model [21], which is neither physically plausible nor empirical, has been modified by Blinn [3] to make it visually more satisfying. Ward [27] has introduced an anisotropic BRDF model that is based on an anisotropic Gaussian micro-facet distribution. Banks [1] has proposed a BRDF which assumes small fibers along the given tangent, resulting in anisotropic reflections.

Hardware accelerated rendering usually uses the Blinn-Phong model, because of its mathematical simplicity. Recently new techniques have been developed to incorporate other more complex BRDF models. Heidrich and Seidel [8] factored the Banks model [1] and the Cook-Torrance model [4] analytically and put the factors into texture maps. Texture mapping was then used to reconstruct the original models. Kautz and McCool [10] first reparameterized BRDFs and then decomposed them numerically. The reconstruction was also done using texture mapping. Both methods assume smoothly varying normals and tangents, i.e. they can neither be used with bump mapping, nor can the parameters of the reflectance model vary.

So far, it has not been possible to achieve interactive per-pixel shading with varying surface and material properties, which our method is capable of.

3 Mapping Reflectance Models to Graphics Hardware

Our method works as illustrated in Figure 1. First, a given analytical BRDF is decomposed into sub-functions, e.g. square root, exponentiation and other complex computations which are not supported by the hardware. We sample these complex functions and put them into texture maps, which will be used later on. In case of Figure 1, we have two two-dimensional texture maps, one for $F(s,t) = \exp \frac{s}{t}$ and one for $G(u,v) = \sqrt{u}^v$.

The parameters of the functions are computed from the surface and material properties, which are allowed to vary per pixel, e.g. the normal, the tangent, the binormal, the surface roughness, anisotropy, and so on. These varying properties are stored in texture maps, which we will call *material textures* from now on. The viewing and light directions are also usually used in the computation of the parameters, but need not to be stored in texture maps, since they only vary smoothly, e.g. they can be specified as the color at the vertices.

The computation of the function parameters is done in the *parameter stage*. For instance, this can be implemented with NVIDIA's register combiner extension [18] or with a combination of multiple passes, blending, and the color matrix of the OpenGL imaging subset [22], see Section 4 for a detailed example. In our example from Figure 1 three parameters have to be computed using dot-products and one is directly read from the texture map.

After the function parameters have been computed, they are used for a dependent texture lookup [1]. The dependent lookup corresponds to evaluating the complex functions which originally could not be computed by the graphics hardware. The results of the dependent texture lookups are combined (e.g. multiplication or addition, depending on how the BRDF was broken up) and we have evaluated the BRDF.

Some reflectance models can be modified so that the dependent texture lookup is not needed at all, making our method work with current hardware, see the next subsections.

The type of reflectance models that can be used with this method is highly dependent on the operations supported by the graphics hardware in the parameter stage. We limit ourselves to models that can be computed with either the register combiners or with multiple passes and the color matrix.

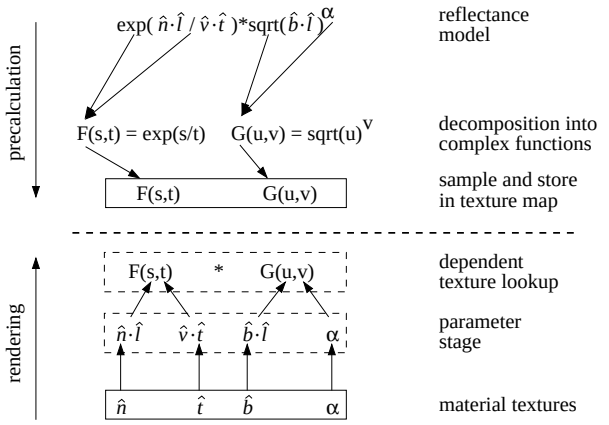

Figure 1: *Example of how our method works conceptually.*

In the rest of this section we will show how our method can be applied to the Banks model, to an anisotropic version of the Blinn-Phong model, and to Ward's model. We also propose an approximation to the Banks model and the Blinn-Phong model so that it can be used without the dependent texture lookup.

Note that we will assume graphics hardware that can compute dot-products in some way and for Ward's model we assume support for two dimensional dependent texture lookups. We will only take a look at the specular part of the BRDFs, since the diffuse reflection can be simply added with a diffuse bump mapping step, see [8, 11] for example. Furthermore, we assume that the surfaces are lit with either a point light source or with a directional light source.

3.1 Banks Model

The first model that we use is the anisotropic Banks model [1] (specular part only):

$$L_o = k_s \cdot (\hat{n} \cdot \hat{l})^+ \left(\sqrt{1-(\hat{v}\cdot\hat{t})^2}\sqrt{1-(\hat{l}\cdot\hat{t})^2} - (\hat{v}\cdot\hat{t})(\hat{l}\cdot\hat{t}) \right)^+$$

where $\hat{v}$ is the global viewing direction, $\hat{l}$ is the global light direction, $\{\hat{n}, \hat{t}, \hat{n} \times \hat{t}\}$ is the local coordinate frame, k_s is the specular coefficient, and $(a)^+$ means that a is clamped to zero, if $a < 0$.

The Banks model is an ad-hoc model, which is not physically based. It assumes small fibers along the given tangent, resulting in anisotropic reflections.

Since this model mainly uses dot-products we only need dependent texture lookups for the square roots. We use the function $R(s) := \sqrt{1-s}$, which we sample in the range $[0,1]$ and put

[1] Dependent texture lookup is currently only available on SGI Octanes in the form of Pixel Textures [23], but is expected to be available on other graphics hardware by the end of the year.

it into a one dimensional texture map. We can rewrite the Banks model using this function:

$$L_o \quad = \quad k_s \cdot (\hat{n} \cdot \hat{l})^+ \left(R\big((\hat{v} \cdot \hat{t})^2\big) R\big((\hat{l} \cdot \hat{t})^2\big) - (\hat{v} \cdot \hat{t})(\hat{l} \cdot \hat{t}) \right)^+ .$$

All the other parts of the reflectance model can be computed using a combination of dot-product computations (using register combiners or the color matrix) and blending operations. Only one two dimensional texture map is needed for storing the tangents and the specular coefficient.

If desired, it is possible to use an approximation of this model in order to make it work on current graphics hardware. The function $R(s)$ can be approximated with $R^*(s) := (1 - s)$, which is a crude approximation, but avoids the dependent texture lookup completely, since it can be computed directly with the graphics hardware. See Section 6 for results using the Banks model.

3.2 Anisotropic Blinn-Phong Model

Blinn [3] modified the original Phong model [21] so that it achieves more realistic reflections:

$$L_o \quad = \quad k_s (\hat{h} \cdot \hat{n})^N,$$

where $\hat{h}$ is the normalized halfway vector between the light and the viewing direction, $\hat{n}$ is the local surface normal, k_s is the specular coefficient, and N controls the sharpness of the highlight. This is the model that is usually used for hardware accelerated bump mapping. We will rewrite the model so that it is anisotropic and that it has more parameters that influence the shape of the highlight.

With Pythagoras' theorem the above dot-product can be expressed the following way:

$$\hat{h} \cdot \hat{n} \quad = \quad \left(\sqrt{\left(1 - (\hat{h} \cdot \hat{t})^2 - (\hat{h} \cdot \hat{b})^2 \right)^+} \right)^N .$$

The model can be made anisotropic by prolonging the tangent and the binormal by a certain amount, which is similar to the anisotropic formulation used by Kindlmann and Weinstein [12]:

$$L_o \quad = \quad k_s \left(\sqrt{\left(1 - \left(\hat{h} \cdot \frac{\hat{t}}{\alpha_x} \right)^2 - \left(\hat{h} \cdot \frac{\hat{b}}{\alpha_y} \right)^2 \right)^+} \right)^N ,$$

where the parameters $\alpha_{x/y} \in [0, 1]$ are used to control the anisotropy and the sharpness of the highlight. The smaller these values are, the sharper the highlight is.

This model uses two functions that are not supported by current graphics hardware: the square root and the exponentiation. These functions can be modeled with one two dimensional dependent texture lookup, where N is used as one parameter and the result of the expression e that is inside the square root is used as the other parameter of the lookup. So we have to sample the function $F(N, e) := (\sqrt{e})^N$ and store it as a texture map. The expression e can be evaluated in the parameter stage using current graphics hardware. We need to store the tangents and the binormals (already divided by α_x respectively α_y) in two two dimensional texture maps. The parameters N and k_s can be stored in the alpha channel of the texture maps.

Since we have an additional set of parameters $\alpha_{x/y}$ that are used to control the sharpness of the highlight, it is not necessary anymore to use the parameter N at all. If desired, we can use the same idea as before and approximate the square root with the identity function. This simplification eliminates the necessity to use dependent texturing and the modified model can thus be used on current graphics hardware. See Section 4 for two rendering algorithms using this model and see Section 6 for some renderings done at interactive rates.

3.3 Ward's Model

Ward [27] has introduced a mathematical simple but physically meaningful BRDF model:

$$f_r := k_s \frac{1}{\sqrt{(\hat{l} \cdot \hat{n})(\hat{v} \cdot \hat{n})}} \frac{1}{4\pi\alpha_x\alpha_y} \cdot \exp\left(-2 \frac{\left(\frac{\hat{h} \cdot \hat{t}}{\alpha_x}\right)^2 + \left(\frac{\hat{h} \cdot \hat{b}}{\alpha_y}\right)^2}{1 + \hat{h} \cdot \hat{n}} \right),$$

where $\hat{h}$ is the halfway vector between the light and the viewing direction, $\{\hat{n}, \hat{t}, \hat{b}\}$ is the local coordinate frame, k_s is the specular coefficient, $\alpha_{x/y}$ control the sharpness of the highlight (the smaller the sharper), and the ratio $\alpha_x : \alpha_y$ controls the anisotropy. The outgoing radiance is computed the following way: $L_o = f_r \cdot (\hat{n} \cdot \hat{l})^+$.

Ward also measured the BRDFs of real materials and fitted his model to the measured data, which are available in [27].

This model needs to be broken up into two complex functions:

$$\begin{aligned} S(p) \quad &:= \quad \frac{1}{\sqrt{p}}, \\ E(m, d) \quad &:= \quad exp(-2\frac{m}{1 + d}), \end{aligned}$$

which leads to the following equation:

$$L_o \quad = \quad \frac{k_s}{4\pi\alpha_x\alpha_y} \cdot S(p) \cdot E(m, d) \cdot (\hat{n} \cdot \hat{l})^+.$$

The two complex functions cannot be computed with graphics hardware, while the other operations are supported. So we sample these functions over their domain and store them in texture maps, which are then used during the dependent texture lookup. The parameters p, m, and d are computed in the parameter stage:

$$\begin{aligned} p \quad &= \quad (\hat{l} \cdot \hat{n})(\hat{v} \cdot \hat{n}), \\ m \quad &= \quad \left(\hat{h} \cdot \frac{\hat{t}}{\alpha_x} \right)^2 + \left(\hat{h} \cdot \frac{\hat{b}}{\alpha_y} \right)^2, \\ d \quad &= \quad \hat{h} \cdot \hat{n}. \end{aligned}$$

It is necessary to store the following data in texture maps: one texture map for $\frac{\hat{t}}{\alpha_x}$, one for $\frac{\hat{b}}{\alpha_y}$, one for $\hat{n}$, and $k_s/(4\pi\alpha_x\alpha_y)$ can for example be stored in the alpha channel of one of the other texture maps. The light, viewing, and halfway vectors are varying smoothly and can be specified per vertex.

3.4 Other models

Obviously this technique can be applied to other BRDF models as long as they can be broken up into functions which depend on a maximum of n parameters, with n being the maximum dimension of the supported dependent texture lookup. Furthermore the graphics hardware must be able to compute the parameters of these functions.

4 Rendering

Here we would like to discuss the algorithm that we used to render surfaces with the anisotropic Blinn-Phong model in order to identify problems when using our method with current graphics hardware. Note that the rendering algorithm for other models is similar.

We will present the rendering algorithm for the simplified Blinn-Phong model, which does not need the dependent texture lookup:

$$L_o = k_s \left(1 - \left(\hat{h} \cdot \frac{\hat{t}}{\alpha_x}\right)^2 - \left(\hat{h} \cdot \frac{\hat{b}}{\alpha_y}\right)^2\right)^+ . \quad (1)$$

As already mentioned before, we need to store the tangent $\frac{\hat{t}}{\alpha_x}$ in one texture map and the binormal $\frac{\hat{b}}{\alpha_y}$ in a second texture map.

These material textures define the reflective properties of the surface. For instance, if a single scratch is to be modeled on a glossy surface, the tangents should be aligned along the scratch and the α_x should be small compared to α_y giving an anisotropic highlight along the scratch. The rest of the surface can be made isotropic by using $\alpha_x = \alpha_y$.

The vectors $\hat{t}$ and $\hat{b}$ can either be specified in a global coordinate system, which means that the surface which this material is applied to has to be known in advance (including geometry, orientation, and position), which has the big disadvantage that these texture maps can only be applied to one specific surface. Or the vectors can be specified in a local coordinate system, which allows to reuse the material textures for different surfaces. But then the viewing and light direction need to be converted into the same local coordinate system before they can be used [11]. Nonetheless we choose to specify the vectors of all material textures in the same local coordinate system to be able to reuse the material textures.

We will now discuss two algorithms, one using the register combiner extension and one using the color matrix of the OpenGL imaging subset.

4.1 Register Combiners

Register combiners offer some limited programmability in the multitexturing stage. The supported features include the computation of dot-products and signed arithmetic [18].

Using these features, it is quite simple to implement the Blinn-Phong model in one pass. For a given polygon, viewer and point light source (or also directional light source) we compute the light and viewing vectors at every vertex. Then we compute the halfway vector for every vertex and transform it into the local coordinate system of the used material texture. At every vertex the local halfway vector is then specified as the color.

Both material textures are loaded once into texture memory using the multitexturing extension [22]. Some attention has be paid to the dynamic range of the stored vectors. Usually all components of a normalized vector are in the range $[-1, 1]$, but in our case the vectors are divided component-wise with the $\alpha_{x/y}$ values, which are less or equal 1, making the possible dynamic range quite large. In order to limit this range, we are not allowing the $\alpha_{x/y}$ to be smaller than 0.25. We scale all the material textures by 0.25 so that the range is again $[-1, 1]$. This way only two bits of precision are lost and rescaling by a factor of 4 is supported by the register combiner extension. Furthermore, we scale and bias the textures so that the range becomes $[0, 1]$ to conform with the allowed range for texture maps (the register combiners map it back to $[-1, 1]$).

Now we can set up the register combiners. In the first stage we compute the dot-product of the vector stored in first texture (contains the tangent divided by α_x) with the interpolated color (the $\hat{h}$ vector), scale it by four, and output it to the first spare register. In the same stage we also compute the dot-product of the second texture (contains the binormal divided by α_y) with the $\hat{h}$ vector, scale it by four, and output it to the second spare register. In the second stage we compute the squares of both dot-products, add them, and output them again to the first spare register. In the final combiner stage we subtract the first spare register from 1. Now we render the polygon and the register combiners evaluate Equation 1 for every pixel.

4.2 Imaging Subset

The imaging subset [22] supports the application of a 4×4 matrix (called color matrix) to each pixel in an image while it is transferred to or from the frame buffer or to texture memory. Using this color matrix it is possible to compute dot-products. Furthermore a color lookup table can be applied to the result of the color matrix computation, which can be used to compute squares for instance.

We will use the color matrix to compute the dot-product between the halfway vector $\hat{h}$ and the tangents/binormals stored in texture maps, and the color lookup table to square the results. Since the color matrix is constant while it is applied to a texture, only dot-products between a constant vector (stored in the color matrix) and vectors stored in a texture map can be computed. Hence it is necessary to use a directional light source, to assume an orthogonal viewer (at least per polygon), and to use flat surfaces only (i.e. same normal at all vertices), or otherwise the halfway vector would vary over the polygon. Again, the halfway vector has to be computed by hand.

The material textures need to be scaled and biased the same way as before, since the values of a texture must be in $[0, 1]$. It is not necessary to limit the smallest $\alpha_{x/y}$ to 0.25 though. In fact, if floating point textures are used, they can become as small as floating point precision allows, since the color matrix also works with floating point precision.

Rendering is a bit more complicated, needing three passes this time. For every polygon, we set up the the color matrix such that it computes the dot-product between the (constant) halfway vector and the vectors stored in the texture map. At the same time the color matrix has to compensate for the scaling and biasing of the texture map. We also define a color lookup table which squares the result of the color matrix computation. Then we load the first texture map (contains the tangent divided by α_x) into the texture memory, which computes the square of the dot-product with $\hat{h}$. We render the polygon with this texture map into the frame buffer. Now the second texture map (contains the binormal divided by α_y) is loaded to texture memory computing the second dot-product and squaring it. We set blending to addition and render the polygon again, which now adds the second squared dot-product on top. Afterwards the subtract blend extension is used to subtract the intermediate result from one. Now Equation 1 has been evaluated for every pixel of the rendered polygon.

5 Hardware Issues

There are a few problems with the current graphics hardware that limit the application of our method. We will discuss these issues with the help of current OpenGL extensions.

Imaging subset. The imaging subset [22] offers a simple way to apply a 4×4 matrix to vectors which are stored in texture maps. This 4×4 matrix can also be used to compute dot-products. The precision is high, since the color matrix uses floating point arithmetic, but in its current implementation no signed results are possible. Moreover, loading textures and applying the color matrix is expensive.

The imaging subset is one way of doing the computations in the parameter stage, but with its limitations it is necessary to assume a directional light source, an orthogonal viewer, and flat surfaces (i.e. same normal at all vertices), which makes reflections from curved objects look unrealistic.

The imaging subset also supports color lookup tables after the color matrix has been applied. This lookup table can be seen as a one dimensional dependent texture lookup. But its application is limited, because the indices for the lookup can only be computed using the color matrix.

Multitexturing. Multitexturing [24] provides a means to combine multiple different textures in one rendering pass. The possible operations usually include multiplication and addition of textures. The current implementations of multitexturing do not support signed arithmetic, making it too limited for our technique (register combiners are an exception). Furthermore most current implementations of multitexturing are limited to two texture maps.

Register combiners. The register combiner extension [18] provides a very wide range of operations that are supported during the multitexturing stage. Those operations include dot-products, multiplications, sums, differences and signed arithmetic.

The only currently available hardware implementing the register combiner extension (NVIDIA GeForce 256 and NVIDIA Quadro) is limited to 8+1 bits of precision and can only compute a limited number of operations during one multitexturing step. The supported range of values is limited to $[-1, 1]$ for all intermediate results. Some support for scaling intermediate results is available, which can be used to handle larger ranges (see rendering algorithm in Section 4.1), but with the limited precision this can only be used to some extent.

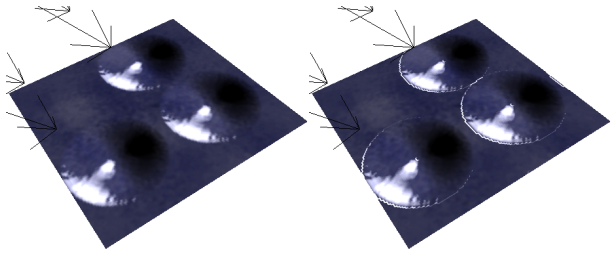

Figure 2: *Bilinear interpolation in the register combiners can cause visible artefacts. Both images were rendered using the anisotropic Blinn-Phong model, the left image using a software renderer and the right image using the register combiners. The additional rings around the bumps (right image) are due to the interpolation.*

Since the register combiner work during the multitexturing stage, the values fed into the register combiners are already filtered bilinearly (if filtering is turned off, textures will look blocky). Bilinear filtering is reasonable for color values, but it causes problems with arbitrary data, such as vectors and material properties. See Figure 2 for an example, where the interpolation of tangent vectors causes visible artefacts. The main problem is that the stored data is interpolated before its used, instead of being used first and then interpolated. A renormalization cube map [11] can be used for interpolation problems with normalized vectors, but this technique cannot be extended to other data types.

Despite these problems, the register combiner provide a very convenient method of computing the parameters in the parameter stage.

Pixel textures. Pixel textures [23], available on SGI Octanes, are the only way of doing dependent texture lookups at the moment. The image that contains the indices has to be written using glDrawPixels, which are used to index into the current active texture map. This is a rather complicated and bandwidth-intensive method to do dependent texture lookups, which needs many copies to and from the host memory, because all intermediate results that are used as indices (parameters of functions in our case) have to be first written to the frame buffer and then read to host memory in order to apply the pixel textures.

5.1 Discussion

A blend of different features would be necessary for a better support of our technique. The register combiners have been proven to be a good way to implement flexible operations in the multitexturing stage. More stages, more concurrent texture maps and higher precision would make them even more useful.

Both rendering algorithms presented in Section 4 have one problem in common. The possibly needed relative viewing, light, and halfway vectors have to be computed by hand at every vertex. This by itself is not ideal, but furthermore it keeps one from using display lists, since these vectors change for every new viewing, light, or object position. Rendering speed could be significantly improved if these values were automatically fed into the multitexturing stage, which has already been proposed in a similar way by McCool and Heidrich [14]. It would also be helpful if these vectors were interpolated using correct spherical interpolation, since the currently supported bilinear interpolation can lead to highlight aliasing across polygon boundaries unless a high tessellation is used.

As seen in Figure 2, the bilinearly interpolated material textures can cause severe visible artefacts. Unfortunately, this is not an easy problem to fix. All calculations (such as dot-products) should work on the stored values *before* they are interpolated. Bilinear interpolation should *then* be applied to the result of the specified computations. Unfortunately, this solution cannot be easily integrated into the multitexturing stage without changing the concept of it.

Dependent texture lookups should be implemented in such a way that an easy use is possible, for example inside the multitexturing/register combiner stage, although an efficient hardware implementation is difficult due to possible memory stalls.

Dependent texture lookups could be avoided in certain cases if the multitexturing stage was able to compute more complex functions like square roots or divisions. Obviously, the set of necessary functions depends on the BRDFs that are to be used, but dot product, square root, division and exponentiation seem to be likely candidates. However, it is unclear if more complex functions should be added or if dependent texture lookups should be used. On the one hand the access pattern of the dependent texture lookup is possibly irregular and therefore resulting in memory stalls but this highly depends on the given bump map. On the other hand the per-pixel computation of complex functions such as divisions is very expensive.

6 Results

Here we would like to show different images that were rendered using our technique. Unless stated otherwise, all the images were rendered in two rendering passes (diffuse and specular pass) on an NVIDIA GeForce 256 using the modified anisotropic Blinn-Phong model (no square root and no exponentiation) at interactive rates (15–25Hz).

Figure 3 shows a single polygon that has scratches along both axes. Depending on the direction of the incoming light, one can see scratches along one axis or the other, or if lit from top, scratches in both directions are visible.

In Figure 4 two spheres are depicted, one which was roughly brushed longitudinally and one latitudinally. One can clearly see how anisotropic highlights occur depending on the orientation of the scratches.

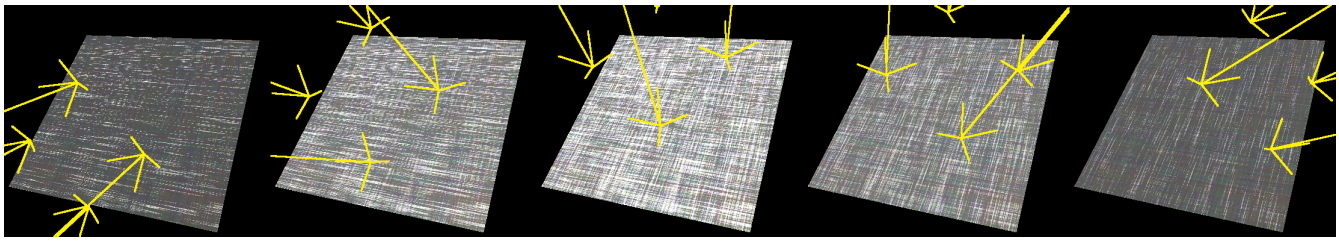

Figure 3: *A single polygon with scratches along both axes, lit from different directions.*

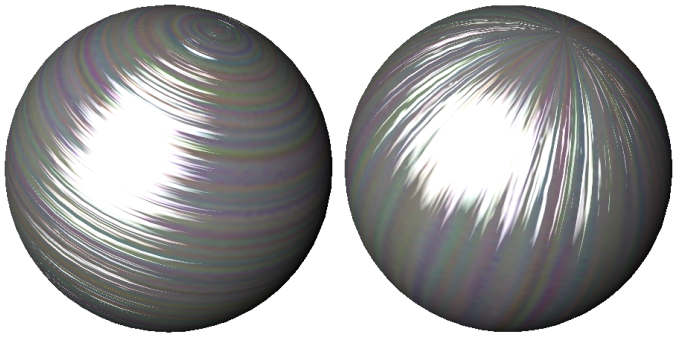

Figure 4: *Two spheres with scratches in different directions. Each sphere consists of 2500 triangles.*

More complex materials can also be rendered using the anisotropic Blinn-Phong model. In Figure 5 you can see a sphere with slightly brushed bumpy gold. In Figure 6 you can see a marble sphere with elevated "veins". The marble itself reflects slightly more light in the darker parts, which was done by increasing α_y in the darker parts. The material textures for the marble can be seen in Figure 7.

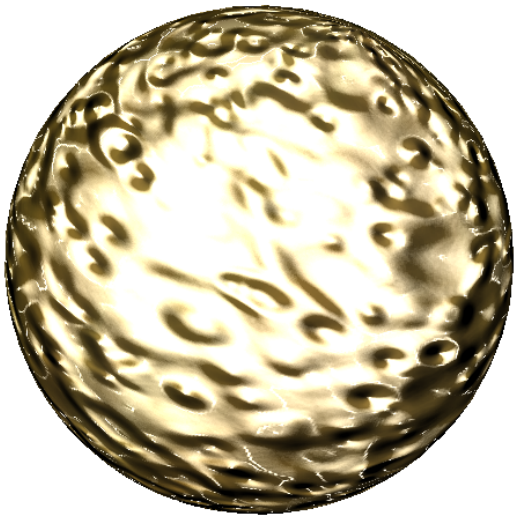

Figure 5: *Gold and bumpy sphere.*

Finally we would like to compare different lighting models. In Figure 8 a polygon with the same slightly brushed bump map was rendered using the Blinn-Phong model, the Banks model, and

Figure 6: *Marble sphere with elevated "veins" using the shift-variant Blinn-Phong model.*

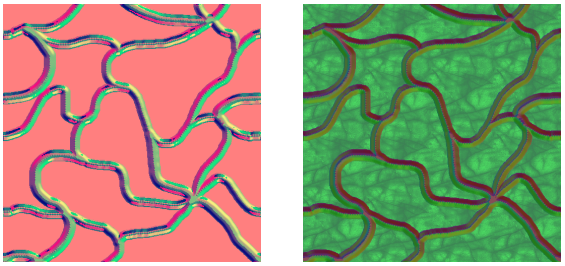

Figure 7: *Material textures for the marble sphere. The left image shows the tangents divided by α_x and the right image shows the binormals divided by α_y.*

Ward's model. The Banks and Ward's model were simulated in software, because of the lacking dependent texturing.

7 Conclusions

We have presented a technique which enables hardware accelerated bump mapping with anisotropic shift-variant BRDFs. This technique allows the local coordinate frame as well as the parameters of the BRDF model to change on a per-pixel basis.

Our method is general enough to handle a wide range of BRDFs, provided that dependent texture lookups are supported. It can still

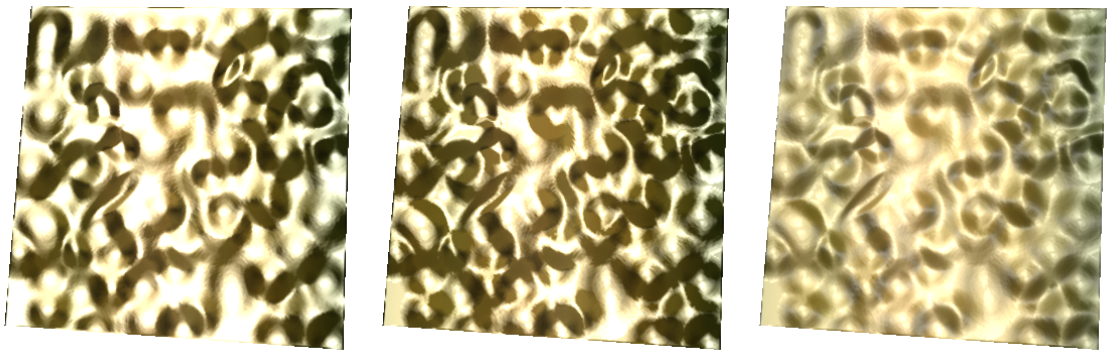

Figure 8: *The same bump mapped surface was rendered with different lighting models. From left to right: Banks model, anisotropic Blinn-Phong, Ward's model.*

be applied to certain BRDFs, even if this feature is not supported. Namely, we propose an approximation to an anisotropic version of the Blinn-Phong model and to the Banks model, which can be then used on current graphics hardware.

We have found that todays graphics hardware has some limitations which may introduce visible artefacts. We have discussed possible enhancements how these limitations could be overcome in future graphics hardware.

There is still a lot of research to be done in this area. Aliasing problems have not been dealt with at all, being still an issue for all bump mapping algorithms. Mip-mapping, a commonly used technique to avoid aliasing artefacts, is already difficult to use with traditional hardware accelerated bump mapping algorithms, but it is especially difficult if complex BRDF models are used, since it is unclear how the material textures should be mip-mapped. A straightforward solution would be to fit a given BRDF model to each pixel of each mip-mapping level, which is not a simple task.

Furthermore it is not clear how the material properties (local coordinate frame, BRDF parameters) can be obtained. Measuring shift-variant BRDFs is generally possible, but requires the measurement of a six dimensional function. It is also conceivable that an artist "draws" these properties using special software, but this software does not exist yet.

8 Acknowledgements

We would like to thank Wolfgang Heidrich for proofreading and discussing the paper.

References

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

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

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

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

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

[6] HAEBERLI, P., AND SEGAL, M. Texture Mapping As A Fundamental Drawing Primitive. In *Fourth Eurographics Workshop on Rendering* (June 1993), Eurographics, pp. 259–266.

[7] HEIDRICH, W. *High-quality Shading and Lighting for Hardware-accelerated Rendering*. PhD thesis, Universität Erlangen-Nürnberg, 1999.

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

[9] HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. In *Symposium on Interactive 3D Graphics* (1999).

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

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

[12] KINDLMANN, G., AND WEINSTEIN, D. Hue-Balls and Lit-Tensors for Direct Volume Rendering of Diffusion Tensor Fields. In *IEEE Visualization '99* (October 1999).

[13] LAFORTUNE, E., FOO, S.-C., TORRANCE, K., AND GREENBERG, D. Non-Linear Approximation of Reflectance Functions. In *Proceedings SIGGRAPH* (Aug. 1997), pp. 117–126.

[14] MCCOOL, M., AND HEIDRICH, W. Texture Shaders. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 1999), pp. 117–126.

[15] MCREYNOLDS, T., BLYTHE, D., GRANTHAM, B., AND NELSON, S. Advanced Graphics Programming Techniques Using OpenGL. In *SIGGRAPH '98 Course Notes* (July 1998).

[16] 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.

[17] NEIDER, J., DAVIS, T., AND WOO, M. *OpenGL - Programming Guide*. Addison-Wesley, 1993.

[18] NVIDIA CORPORATION. *NVIDIA OpenGL Extension Specifications*, Oct. 1999. 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] PEERCY, M., AIREY, J., AND CABRAL, B. Efficient Bump Mapping Hardware. In *Proceedings SIGGRAPH* (Aug. 1997), pp. 303–306.

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

[22] SEGAL, M., AND AKELEY, K. *The OpenGL Graphics System: A Specification (Version 1.2.1)*, 1999.

[23] SILICON GRAPHICS INC. *Pixel Texture Extension*, Dec. 1996. Specification document, available from http://www.opengl.org.

[24] SILICON GRAPHICS INC. *Multitexture Extension*, Sept. 1997. Specification document, available from http://www.opengl.org.

[25] 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.

[26] VOORHIES, D., AND FORAN, J. Reflection Vector Shading Hardware. In *Proceedings SIGGRAPH* (July 1994), pp. 163–166.

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

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