A Practical and Robust Bump-mapping Technique for Today's GPUs

Mark J. Kilgard
NVIDIA Corporation
3535 Monroe Street
Santa Clara, CA 95051
(408) 615-2500
mjk@nvidia.com

July 5, 2000

ABSTRACT

Bump mapping is a normal-perturbation rendering technique for simulating lighting effects caused by patterned irregularities on otherwise locally smooth surfaces. By encoding such surface patterns in texture maps, texture-based bump mapping simulates a surface's irregular lighting appearance without modeling the patterns as true geometric perturbations to the surface. Bump mapping is advantageous because it can decouple a texture-based description of small-scale surface irregularities used for per-pixel lighting computations from the vertex-based description of large-scale object shape required for efficient transformation and rasterization. This paper describes a practical and robust bump-mapping technique suited for the capabilities of today's Graphics Processor Units (GPUs).

This paper has six sections including the introduction. The second section presents the theory of bump mapping. The third section reviews several existing hardware bump-mapping techniques. The fourth section describes the cube map texturing and "register combiners" features of NVIDIA's GeForce 256 and Quadro GPUs. These features in combination are the basis for the bump mapping technique presented in the fifth section. This technique achieves real-time performance on current generation NVIDIA GPUs. The technique can model ambient, diffuse, and specular lighting terms, a textured surface decal, and attenuated and spotlight illumination. The technique is notable because of its robustness and relative fidelity to Blinn's original mathematical formulation of bump mapping. The sixth section concludes. Several appendices provide OpenGL implementation details.

Keywords

Bump mapping, texturing, per-pixel lighting, register combiners, GeForce 256, Quadro, OpenGL.

1 INTRODUCTION

Video games use texture-mapping hardware to generate interactive computer-generated imagery with astounding levels of per-pixel detail, yet conventional hardware texture mapping does not give rendered surfaces a "textured" feel in the everyday sense of the term. Conventional hardware texture mapping gives polygonal models a colorful yet flat appearance. Even when textured models are augmented by per-vertex lighting computations, polygonal flatness is still evident. The problem is that what computer graphics practitioners call texture is not what people mean by the word texture in everyday usage.

In everyday usage the word "texture" refers to bumps, wrinkles, grooves, or other irregularities on surfaces. The computer graphics notion of a texture is more akin to the everyday notion of a decal. In the everyday world, we recognize bumpy surfaces because of the way light interacts with the surface irregularities. In a strict sense, these bumps and irregularities are part of the complete geometric form of everyday objects, but in a looser sense, the scale of these irregularities is quite small relative the overall geometric form of the object. This is the reason that people are comfortable calling a stucco wall "flat" despite its stucco texture and consider a golf ball "round" despite its dimples.

To capture the everyday sense of texture, computer graphics practitioners use an extension of texture mapping known as bump mapping. Blinn invented bump mapping in 1978 [4]. Bump mapping is a texture-based rendering approach for simulating lighting effects caused by patterned irregularities on otherwise locally smooth surfaces. By encoding such surface patterns in texture maps, bump mapping simulates a surface's irregular lighting appearance without the complexity and expense of modeling the patterns as true geometric perturbations to the surface.

Rendering bumpy surfaces as actual geometric perturbations is just too computationally expensive and data intensive. Moreover, the scale of the minute geometric displacements is typically less than the discrete pixel resolution used for rendering which leads to aliasing issues. Bump mapping is a more efficient approach because it decouples the texture-based description of small-scale surface irregularities used for per-pixel lighting computations from the vertex-based description of

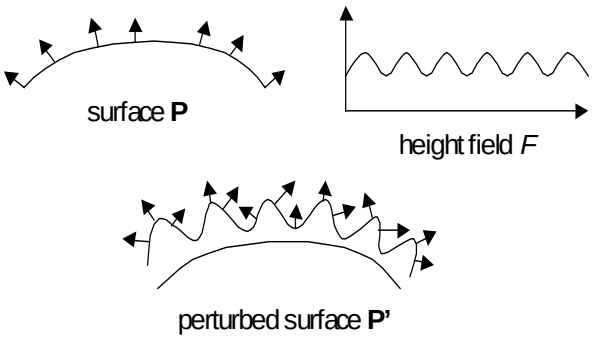

Figure 1. A surface perturbed by a height field makes a bumpy surface.

large-scale object shape required for efficient transformation, rasterization, and hidden surface removal.

Still the computations required for bump mapping as originally formulated by Blinn are considerably more expensive than those required for conventional hardware texture mapping. Many attempts have been made over the last two decades to reformulate bump mapping into a form suitable for hardware implementation. In general, these attempts suffer from various and often serious limitations.

This paper describes a new practical and robust bump-mapping technique suited to NVIDIA's current generation Graphics Processing Units (GPUs). This paper also compares and contrasts the new technique with other hardware bump mapping techniques described in the literature.

This paper has six sections. After this introduction, the second section presents the mathematics of bump mapping. The third section reviews several hardware bump-mapping techniques described in the literature. The fourth section describes the cube map texturing and "register combiners" features of NVIDIA's GeForce 256 and Quadro GPUs. These features in combination with multitexturing are the basis for the bump mapping technique presented in the fifth section. This technique achieves real-time performance on NVIDIA GPUs. The technique can model ambient, diffuse, and specular lighting terms, a textured surface decal, and attenuated and spotlight illumination. The technique supports both a locally positioned light and a local viewer model. The technique is notable because of its robustness and relative fidelity to Blinn's original mathematical formulation of bump mapping. The sixth section concludes. Several appendices provide OpenGL implementation details.

2 BUMP MAPPING MATHEMATICS

Bump mapping is divided into two tasks. First, a perturbed surface normal is computed. Then, a lighting computation is performed using the perturbed normal.

These two tasks must be performed at each and every visible fragment of a bump-mapped surface. Because we desire a hardware solution that runs at real-time rates, computational efficiency is an imperative.

2.1 Classic Height Field Bump Mapping

The classic formulation of bump mapping developed by Blinn computes perturbed surface normals for a parametric surface as if a height field displaced the surface in the direction of the unperturbed surface normal. The surface, without any actual displacement, is then rendered and illuminated based on the perturbed surface normals. These computations are performed at each and every visible pixel on the surface.

The discussion in this and the next two subsections restates Blinn's classic motivation for bump mapping [4].

Assume a surface is defined by a bivariate vector function $\boldsymbol{P}(u,v)$ that generates 3D points (x,y,z) on the surface. (A bivariate vector function is a function of two variables where the result is a vector. A 2D texture map containing colors represented as RGB triples is one example of a bivariate vector function.) The surface normal for a point on **P** is defined as

$$\mathbf{N}(u,v) = \frac{\partial \mathbf{P}(u,v)}{\partial u} \times \frac{\partial \mathbf{P}(u,v)}{\partial v}$$

Equation 1

A second bivariate scalar function $F(u,v)$ defines displacement magnitudes from the surface $\mathbf{P}(u,v)$. The displacement direction is determined by normalizing $\mathbf{N}(u,v)$. F can be interpreted geometrically as a height field that displaces the surface **P**. Together, **P** and F define a new displaced surface $\mathbf{P'}(u,v)$ as shown in Figure 1 and defined as follows

$$\mathbf{P'}(u,v) = \mathbf{P}(u,v) + F(u,v)\frac{\mathbf{N}(u,v)}{|\mathbf{N}(u,v)|}$$

The surface normal for a point on **P'** is defined as

$$\mathbf{N'}(u,v) = \frac{\partial \mathbf{P'}(u,v)}{\partial u} \times \frac{\partial \mathbf{P'}(u,v)}{\partial v}$$

Equation 2

Expanding these partial derivatives of **P'** with the chain rule results in

$$\frac{\partial \mathbf{P'}}{\partial u} = \frac{\partial \mathbf{P}}{\partial u} + \frac{\partial F}{\partial u}\left(\frac{\mathbf{N}}{|\mathbf{N}|}\right) + F\left(\frac{\partial \dfrac{\mathbf{N}}{|\mathbf{N}|}}{\partial u}\right)$$

Equation 3

$$\frac{\partial \mathbf{P'}}{\partial v} = \frac{\partial \mathbf{P}}{\partial v} + \frac{\partial F}{\partial v}\left(\frac{\mathbf{N}}{|\mathbf{N}|}\right) + F\left(\frac{\partial \dfrac{\mathbf{N}}{|\mathbf{N}|}}{\partial v}\right)$$

Equation 4

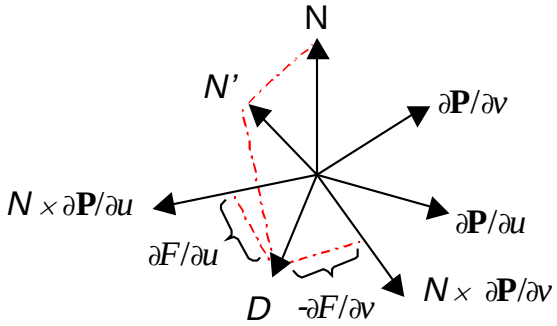

Figure 2. Interpreting a perturbation as an offset vector.

In the context of bump mapping, F represents the bumpiness of the surface. This bumpiness can be considered micro-displacements that are negligible relative to the overall scale of the surface **P**. So assuming the magnitude of F is negligible, the rightmost term in both Equation 3 and Equation 4 can be approximated by zero. Note that even if F itself is negligible, the partial derivatives of F are significant.

Expanding the cross product in Equation 2 gives the following expression for **N'**

$$\mathbf{N'} = \left(\frac{\partial \mathbf{P}}{\partial u} + \frac{\partial F}{\partial u}\left(\frac{\mathbf{N}}{|\mathbf{N}|}\right)\right) \times \left(\frac{\partial \mathbf{P}}{\partial v} + \frac{\partial F}{\partial v}\left(\frac{\mathbf{N}}{|\mathbf{N}|}\right)\right)$$

which evaluates to

$$\mathbf{N'} = \frac{\partial \mathbf{P}}{\partial u} \times \frac{\partial \mathbf{P}}{\partial v} + \frac{\frac{\partial F}{\partial u}\left(\mathbf{N} \times \frac{\partial \mathbf{P}}{\partial v}\right)}{|\mathbf{N}|} + \frac{\frac{\partial F}{\partial v}\left(\frac{\partial \mathbf{P}}{\partial u} \times \mathbf{N}\right)}{|\mathbf{N}|} + \frac{\frac{\partial F}{\partial u}\frac{\partial F}{\partial v}(\mathbf{N} \times \mathbf{N})}{|\mathbf{N}|^2}$$

The first term is just **N** based on Equation 1. The last term is zero because $\mathbf{N} \times \mathbf{N} = 0$. Therefore, **N'** simplifies to

$$\mathbf{N'} = \mathbf{N} + \frac{\frac{\partial F}{\partial u}\left(\mathbf{N} \times \frac{\partial \mathbf{P}}{\partial v}\right) - \frac{\partial F}{\partial v}\left(\mathbf{N} \times \frac{\partial \mathbf{P}}{\partial u}\right)}{|\mathbf{N}|}$$

Equation 5

When analytical partial derivatives are not available for **P** and F, the partial derivatives in Equation 5 can be approximated using finite differences.

Equation 5 computes the perturbed surface normal required for bump mapping, but the computations required are too expensive for direct per-fragment evaluation at hardware rendering rates. Additionally, the normal vector as used in lighting equations must be normalized, so **N'** requires normalization. More efficient approaches to compute normalized **N'** are required.

Blinn gives two different geometric interpretations of **N'**. These two interpretations lead to two different ways to represent normal perturbations in a bump map. Blinn's first interpretation leads to encoding bump maps as offset vectors. Blinn's second interpretation leads to encoding bump maps as vector rotations. These two interpretations serve as the basis for a wide variety of bump-mapping techniques.

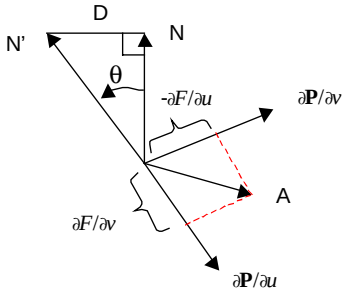

Figure 3. Interpreting a perturbation as a rotation.

2.2 Offset Vector Bump Maps

Blinn's first geometric interpretation of Equation 5 notes that **N'** can be viewed as **N** offset by a vector **D**. That is

$$\mathbf{N'} = \mathbf{N} + \mathbf{D}$$

Equation 6

where

$$\mathbf{D} = \frac{\frac{\partial F}{\partial u}\left(\mathbf{N} \times \frac{\partial \mathbf{P}}{\partial v}\right) - \frac{\partial F}{\partial v}\left(\mathbf{N} \times \frac{\partial \mathbf{P}}{\partial u}\right)}{|\mathbf{N}|}$$

Equation 7

Blinn gives a geometric interpretation of **D**. The two vectors $\mathbf{N} \times \partial\mathbf{P}/\partial v$ and $\mathbf{N} \times \partial\mathbf{P}/\partial u$ reside in the tangent plane of the surface. Scaling each vector by an amount proportional to the u and v partial derivatives of F and adding the two vectors produces **D**. The vector **D** plus the unperturbed vector **N** produce **N'**. This interpretation is shown in Figure 2.

This implies that another way to represent a bump map in addition to a height field is to store the offset vector function $\mathbf{D}(u,v)$. A bump map texture that represents normal perturbations this way is called an *offset vector bump map* or simply an *offset map*.

2.3 Vector Rotation Bump Maps

Blinn's second geometric interpretation of Equation 5 treats the normal perturbation as vector rotation. **N'** can be generated by rotating **N** along an axis in the tangent plane of the surface. This axis vector is the cross product of **N'** and **N**. Because **N**, **N'**, and **D** are all in the same plane

$$\mathbf{N} \times \mathbf{N'} = \mathbf{N} \times (\mathbf{N} + \mathbf{D}) = \mathbf{N} \times \mathbf{D}$$

Expanding **D** in the $\mathbf{N} \times \mathbf{D}$ expression above using Equation 7 leads to

$$\mathbf{N}\times\mathbf{N'}=\frac{\frac{\partial F}{\partial u}\left(\mathbf{N}\times\left(\mathbf{N}\times\frac{\partial \mathbf{P}}{\partial v}\right)\right)-\frac{\partial F}{\partial v}\left(\mathbf{N}\times\left(\mathbf{N}\times\frac{\partial \mathbf{P}}{\partial u}\right)\right)}{|\mathbf{N}|}$$

This can be rewritten using the vector identity $\mathbf{Q}\times(\mathbf{R}\times\mathbf{S}) = \mathbf{R}(\mathbf{Q}\bullet\mathbf{S}) - \mathbf{S}(\mathbf{Q}\bullet\mathbf{R})$ and the facts that $\mathbf{N}\bullet\partial\mathbf{P}/\partial v = \mathbf{N}\bullet\partial\mathbf{P}/\partial u = 0$ because of Equation 1 and that $\mathbf{N}\bullet\mathbf{N}=|\mathbf{N}|^2$. The simplifications lead to

$$\mathbf{N}\times\mathbf{N'}=|\mathbf{N}|\left(\frac{\partial F}{\partial v}\frac{\partial \mathbf{P}}{\partial u}-\frac{\partial F}{\partial u}\frac{\partial \mathbf{P}}{\partial v}\right)=|\mathbf{N}|\mathbf{A}$$

Equation 8

where

$$\mathbf{A}=\frac{\partial F}{\partial v}\frac{\partial \mathbf{P}}{\partial u}-\frac{\partial F}{\partial u}\frac{\partial \mathbf{P}}{\partial v}$$

The vector **A** is perpendicular to the gradient vector of *F*, i.e. $(\partial F/\partial u, \partial F/\partial v)$, when expressed in the tangent plane coordinate system with basis vectors $\partial\mathbf{P}/\partial v$ and $\partial\mathbf{P}/\partial u$. This means **A** acts as an axis that tips **N** "downhill" due to the sloping (i.e., the gradient) of *F*. So the perturbed surface normal **N'** is the result of a rotation around the **A** axis.

The next task is to determine the angle of rotation. Since $|\mathbf{N}|\mathbf{A}=\mathbf{N}\times\mathbf{D}$ by Equation 8 and **N** is perpendicular to **D** then

$$|\mathbf{N}\times\mathbf{D}|=|\mathbf{N}||\mathbf{D}|$$

So Equation 8 implies that

$$|\mathbf{D}|=|\mathbf{A}|$$

By noting that **N**, **D**, and **N'** form a right triangle, the angle of the perturbation rotation around **A** is θ where

$$\tan(\theta)=\frac{|\mathbf{N}|}{|\mathbf{D}|}=\frac{|\mathbf{N}|}{|\mathbf{A}|}$$

This is illustrated in Figure 3.

So in addition to height fields and offset vector maps, a third way to store a bump map is encoding normal perturbations as a (u,v) map of vector rotations. Euler angles, 3x3 rotation matrices, and quaternions are all reasonable representations for vector rotations. A bump map texture that represents normal perturbations this way is called a *normal perturbation map* or simply a *normal map*.

If $\mathrm{R}(u,v)$ is such a normal perturbation map, then

$$\frac{\mathbf{N'}}{|\mathbf{N}|}=\mathbf{N}(u,v)\otimes\mathrm{R}(u,v)$$

Equation 9

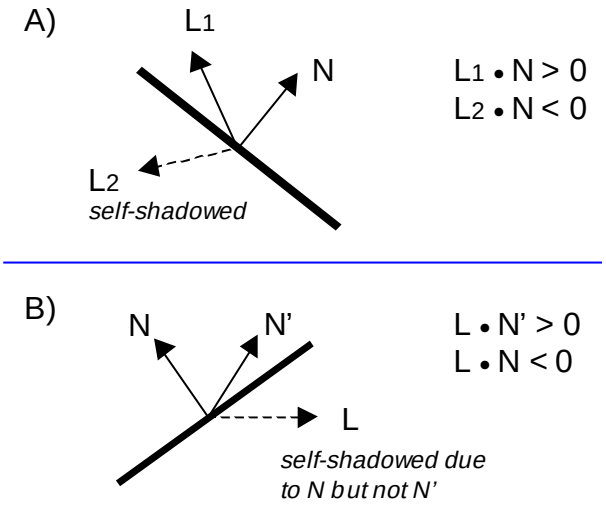

Figure 4. Self-shadowing situations.

After a perturbed normal is computed for a given fragment, the perturbed normal and other lighting parameters are fed to a per-fragment lighting model.

Because rotations do not change the length of a vector, one consequence of Equation 9 is that the resulting normal is normalized and therefore immediately usable in lighting computations.

2.4 Bumped Diffuse Lighting

Diffuse lighting interactions are typically modeled with a lighting model based on Lambert's law. This pseudo-law says that surfaces uniformly reflect incident light. The perceived intensity of a surface that obeys Lambert's law and that is illuminated by a single point light source is computed as

$$I_{lambert}=I_{ambient}+k_{diffuse}\max(0,\mathbf{L}\bullet\mathbf{N})$$

Equation 10

$I_{ambient}$ is the ambient perceived intensity, $k_{diffuse}$ is the percentage of reflected diffuse light, **L** is the normalized light vector and **N** is the normalized surface normal vector.

When discussing lighting equations, the vectors discussed are assumed to be normalized unless otherwise noted.

The subexpression $\max(0,\mathbf{L}\bullet\mathbf{N})$ forces the light's perceived diffuse intensity to zero if $\mathbf{L}\bullet\mathbf{N}$ is negative. This clamping corresponds to the situation when the light and the viewer are on opposite sides of the surface tangent plane. This is shown in Figure 4-A. In this situation, the point on the surface is said to "self shadow." Clamping also avoids the nonsensical contribution of negative reflected light.

2.4.1 Bumped Diffuse Self-shadowing

When bump mapping, there are really two surface normals that should be considered for lighting. The unperturbed normal **N** is based on the surface's large-scale modeled geometry while the perturbed **N'** is based on the surface's small-scale micro-structure.

Either normal can create self-shadowing situations. Figure 4-B shows a situation where the perturbed normal is subject to illumination based the light direction, i.e. when **L**•**N'** is positive. However, the point on the surface arguably should not receive illumination from the light because the unperturbed normal indicates the point is self-shadowed due to the surface's large-scale geometry.

To account for self-shadowing due to either the perturbed or unperturbed surface normal, Equation 10 can be rewritten as

$$I_{lambert} = I_{ambient} + k_{diffuse}\ s_{self}\ \max(0, \mathbf{L} \bullet \mathbf{N'})$$

Equation 11

where

$$s_{self} = \begin{cases} 1, & \mathbf{L} \bullet \mathbf{N} > 0 \\ 0, & \mathbf{L} \bullet \mathbf{N} \le 0 \end{cases}$$

Without this extra level of clamping, bump-mapped surfaces can show illumination in regions of the surface that should be self-shadowed based on the vertex-specified geometry of the model.

In practice, the step function definition of s_{self} shown above leads to temporal aliasing artifacts. During animation, pixels along the geometric self-shadowing boundary of bump-mapped surfaces may pop on and wink off abruptly. Modifying s_{self} to make a less abrupt transition can minimize this aliasing artifact. For example:

$$s_{self} = \begin{cases} 1, & \mathbf{L} \bullet \mathbf{N} > c \\ \frac{1}{c}(\mathbf{L} \bullet \mathbf{N}), & 0 < \mathbf{L} \bullet \mathbf{N} \le c \\ 0, & \mathbf{L} \bullet \mathbf{N} \le 0 \end{cases}$$

Equation 12

This version of s_{self} uses a steep linear ramp to transition between 0 and 1. When c=0.125, this modified s_{self} is effective at minimizing popping and winking artifacts and is straightforward for hardware implementation.

2.4.2 Bump Map Filtering for Diffuse Lighting

Keep in mind that Equation 11 assumes that the **L** and **N'** vectors are normalized to unit length. The equation also assumes a single unperturbed normal and a single perturbed surface normal. In practice, a pixel's bump map footprint covers a region of the bump map. Assuming a discrete bump map representation, the pixel's intensity should be an average of Equation 11 over the collection of perturbed normals within the pixel's bump map footprint. Assuming a discrete collection of n perturbed normals within a given pixel footprint and equal weighting of the samples, Equation 11 becomes

$$I_{lambert} = I_{ambient} + \frac{1}{n}\sum_{i=1}^{n}\left(k_{diffuse}\ s_{self}\ \max(0, \mathbf{L} \bullet \mathbf{N'}_i)\right)$$

Equation 13

In conventional texture mapping, pre-filtering of the discrete samples within a texture map is applicable because the texture's color contribution can be factored out of the shading computation. Such an approach is not immediately applicable to Equation 11 because the subexpression max(0,**L**•**N'**) is not a linear expression.

Unfortunately, direct application of Equation 13 is impractical for interactive hardware rendering so some pre-filtering scheme for bump maps is required.

The reason that the subexpression max(0,**L**•**N'**) cannot be factored out is solely due to the self-shadow clamping. If we simply ignore self-shadow clamping and assume that **L** and s_{self} are uniform over the pixel's footprint in bump map space, then

$$\frac{1}{n}\sum_{i=1}^{n}(K_{diffuse}\, s_{self}\ \mathbf{L} \bullet \mathbf{N'}_i) = k_{diffuse}\, s_{self}\, \mathbf{L} \bullet \left(\frac{1}{n}\sum_{i=1}^{n}\mathbf{N'}_i\right)$$

Assuming that **L** and s_{self} are uniform implies that **N** is uniform. Hardware renderers only have a single **N** vector available per-pixel because **N** is interpolated at pixel resolution so assuming that **N** is uniform over a pixel is a practical assumption. **L** can also assumed to be uniform over the pixel if the light is directional. Even in the case of a positional light, the light would have to be extremely close to the surface for **L** to vary significantly.

Arguably self-shadowing of individual perturbed normals can be safely ignored under many circumstances if we simply clamp to zero the dot product of **L** with the averaged **N'**. This assumption correctly handles two very important cases. These two cases are when all and when none of the perturbed normals are self-shadowed. A situation where the assumption is very poor requires an extreme distribution of perturbed surface normals that is unlikely if **N'** is generated from a height field with adequate resolution.

Appreciating the assumptions suggested above, a reasonable bump map pre-filtering scheme for a diffuse lighting model is

$$\mathbf{N'}_{filtered} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{N'}_i$$

Equation 14

Then the perceived intensity of the surface is computed as

$$I_{lambert} = I_{ambient} + k_{diffuse}\ s_{self}\ \max(0, \mathbf{L} \bullet \mathbf{N'}_{filtered})$$

Equation 15

Equation 15 matches Equation 11 except that **N'** is replaced by $\mathbf{N'}_{\mathbf{filtered}}$. Note that $\mathbf{N'}_{\mathbf{filtered}}$ is not generally normalized. Renormalizing $\mathbf{N'}_{\mathbf{filtered}}$ might seem reasonable and has been suggested in the literature [21], but *not* renormalizing when computing a surface's perceived diffuse intensity is actually more correct.

Consider the dimpled surface of a golf ball viewed from a distance (assume golf balls are Lambertian for a moment). Also consider a smooth Lambertian ball of the same size viewed from the same distance. Due to its dimpled surface, the perceived intensity of the golf ball should be dimmer than the perceived intensity of the smooth ball.

The brute force filtering in Equation 13 correctly models this dimming of bumpy diffuse surfaces. A robust bump map pre-filtering scheme should also reproduce this effect.

So consider what happens when Equation 14 is used to filter a bump map. If the bump map is indeed "bumpy," a filtered collection of sampled perturbed normals will point in varying directions though all will be unit length. Averaging these varied normals produces a vector with a length that must be less than one. Similarly, consider what happens when Equation 14 is used to filter a "flat" bump map. A flat bump map implies that $\mathbf{N}=\mathbf{N'}=\mathbf{N'}_{\mathbf{filtered}}$ so $\mathbf{N'}_{\mathbf{filtered}}$ is unit length everywhere.

Under otherwise identical circumstances, the filtered "bumpy" bump map correctly produces a dimmer perceived intensity than the filtered "flat" bump map.

Note that if $\mathbf{N'}_{\mathbf{filtered}}$ is renormalized then bumpy diffuse surfaces incorrectly appear too bright.

This result suggests that linear averaging of normalized perturbed normals is a reasonable filtering approach for the purpose of diffuse illumination. It also suggests that if bump maps are represented as perturbed normal maps then such normal maps can be reasonably filtered with conventional linear-mipmap-linear texturing hardware when used for diffuse lighting.

2.5 Bumped Specular Lighting

Bump-mapped specular lighting interactions are typically modeled with either the Blinn or Phong lighting models. Both are empirical models.

The Blinn model [3] for a single point light source is

$$\begin{aligned} I_{ambient} &= I_{ambient} \\ &\quad + k_{diffuse} \max(0, \mathbf{L} \bullet \mathbf{N}) \\ &\quad + k_{specular}\ s_{self}\ \max(0, \mathbf{H} \bullet \mathbf{N})^{shininess} \end{aligned}$$

Equation 16

where **H** is the half-angle vector defined as

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|}$$

V is the normalized view vector (the vector from the surface position to the eye position).

The Phong model [20] for a single point light source is

$$\begin{aligned} I_{ambient} &= I_{ambient} \\ &\quad + k_{diffuse} \max(0, \mathbf{L} \bullet \mathbf{N}) \\ &\quad + k_{specular}\ s_{self}\ \max\left(0, \mathbf{L} \bullet \frac{\mathbf{R}}{|\mathbf{R}|}\right)^{shininess} \end{aligned}$$

Equation 17

where **R** is the reflected view vector defined as

$$\mathbf{R} = \mathbf{V} - 2\mathbf{N}^T(\mathbf{N} \bullet \mathbf{V})$$

Equation 18

Both models assume the vectors **N**, **L**, and **V** are normalized. Both models use exponentiation by a shininess exponent to model specular highlights. Both models use s_{self} to avoid the specular contribution from contribution when the light is self-shadowed.

2.5.1 Blinn Bumped Specular Lighting

Blinn's original bump-mapping scheme assumes the Blinn lighting model. The Blinn model is both less expensive to compute and more physically plausible than the Phong model. Moreover, the half-angle vector is independent of **N** so substituting a perturbed normal **N'** into the lighting model is straightforward.

The Blinn lighting model assumes that **N** (or **N'** when bump mapping) is normalized. For this reason, the Blinn model works well in conjunction with a bump map encoded as a normal map because such maps generate normalized perturbed normals (the rotation of a normalized vector remains normalized). However, using the Blinn model with an offset map requires a per-pixel renormalization.

The Blinn model similarly assumes that **L** and **H** are normalized per-pixel.

When computing **H•N'**, arguably for self-shadowing reasons, the specular contribution should be forced to zero if the specular contribution computed with the unperturbed normal, i.e. **H•N**, goes negative. This effect can be folded into the specular version of the s_{self} term for Equation 16.

2.5.2 Phong Bumped Specular Lighting

The Phong model has several interesting advantages despite its additional complexity. The reflection vector **R** is a function of the surface normal so the Phong model requires a per-pixel reflection vector computation. However, once the reflection vector is computed, it can be used to access a cube map texture encoding the specular illumination.

Accessing a cube map has three noteworthy advantages. First, the cube map avoids the need to renormalize **R** per-pixel. Second, the exponentiation can be encoded into the cube map. Moreover, the object's filtered environment can be encoded into the cube map to support so-called "bumped environment mapping." Third, as noted by Voorhies [27], the normal vector **N** need not be normalized if Equation 18 is rewritten to multiply both terms by the length of the unnormalized normal vector squared. R then can be computed as

$$\mathbf{R} = \mathbf{V}(\mathbf{N} \bullet \mathbf{N}) - 2\mathbf{N}^T(\mathbf{N} \bullet \mathbf{V})$$

where **N** is not necessarily normalized. This approach works well when the bump map is encoded as an offset map because such maps do not guarantee normalized perturbed normals (refer back to Equation 6).

The advantages of the Phong model for bump mapping should not be overstated though. The technique requires a bump map texture access *and* a subsequent dependent cube map texture access. When implemented in hardware, expensive latency fifos

are needed to effectively hide the memory latency of the second dependent texture access. Because the perturbed normals and the specular cube map will not be oriented in the same coordinate system, a normal coordinate system transformation is required. Ideally, this transform should be updated per-pixel to avoid artifacts. The specular cube map assumes the specular illumination comes from an infinite environment. This means local lights, attenuation, and spotlight effects are not possible though an unlimited number of directional light sources can be encoded into a specular cube map. Good pre-filtering of a specular cube map, particularly for dull surfaces, is expensive. Likewise, changing the lighting configuration or environment requires regenerating the cube map. Each material with a unique shininess exponent requires a differently pre-filtered specular cube map. Saving the expense of renormalization of the perturbed normal is only a true advantage if the diffuse illumination can also be computed with an unnormalized perturbed normal too. The obvious way to accomplish this is with a second diffuse cube map, particularly if the specular cube map encodes multiple directional lights. Unfortunately, a diffuse cube map requires an additional dependent cube map texture fetch unit or requires a second rendering pass.

2.5.3 Bump Map Filtering for Specular Lighting

Both models compute the diffuse contribution based on a Lambertian model identical to the previous subsection. Therefore, the previous analysis to justify pre-filtering of perturbed normals for the diffuse contribution still applies.

However, the exponentiation in the specular contribution makes a similar attempt to factor **L** outside the specular dot product rather dubious. Assuming a discrete collection of *n* perturbed normals within a given pixel footprint and equal weighting of the samples, the perceived specular intensity should be

$$I_{specular} = \frac{1}{n}\sum_{i=1}^{n} \max\left(0, H \bullet N_i\right)^{shininess}$$

Equation 19

The conventional interpretation of the exponentiation is that it models an isotropic Gaussian micro-distribution of normals. The exponent is often referred to as a measure of the surface's roughness or shininess. In the context of bump mapping, this supposes some statistical bumpiness on a scale smaller than even the bump map.

Fournier [10] proposed a pre-filtering method for normal maps that substitutes a small set of "Phong peaks" from a larger sample of perturbed normals to be filtered. The perceived specular contribution is then approximated as a weighted sum of the smaller number of Phong peaks. The approximation is

$$\frac{1}{n}\sum_{i=1}^{n} \max\left(0, \mathbf{H} \bullet \mathbf{N}_i\right)^{shininess} \cong \sum_{j=1}^{m} w_j \max\left(\mathbf{H} \bullet \mathbf{N}_j\right)^{e_j}$$

Equation 20

where *m* is the number of Phong peaks, w_j is the weighting for peak *j*, $\mathbf{N}_j$ is the normalized direction of peak *j*, and e_j is the exponent for peak *j*. Fournier suggests fitting the Phong peaks using an expensive non-linear least-squares approach.

Fournier's approach is poorly suited for hardware implementation. Each peak consists of at least four parameters: a weight, an exponent, and a direction (θ,ϕ). Fournier also suggests that as many as seven peaks may be required for adequate reconstruction and multiple sets of peaks may need to be averaged for each pixel.

Schilling [23] presented another approach that is more amenable to hardware implementation yet still out of reach for today's available hardware resources. Schilling proposes the construction of a roughness map that encodes the statistical covariance of perturbed normals within a given pixel footprint. Schilling's approach proposes a considerably more tractable representation of the normal distribution than Fournier's more expensive and comprehensive approach.

The covariance provides enough information to model anisotropic reflection effects. During rendering, information from the roughness map is used for anisotropic filtering of a specular cube map or as inputs to a modified Blinn lighting model. A more compact roughness scheme is possible by limiting the roughness map to isotropic roughness.

Without a tractable approach to pre-filtering bump maps for specular illumination given today's available hardware resources, we are left with few options. One tractable though still expensive option is simply to apply Equation 19 for some fixed number of samples using multiple passes, perhaps using an accumulation buffer to accumulate and weight each distinct perturbed normal's specular contribution.

With no more effective option available, we again consider pre-filtering the bump map by simply averaging the perturbed normals within each pixel's footprint and renormalizing. This is precisely what we decided was incorrect for diffuse illumination, but in the case of specular illumination, evaluating Equation 16 or Equation 17 without a normalized normal is something to be avoided to have any chance of a bright specular highlight. We can at least observe that this approach is the degenerate case of Equation 20 where *m* equals 1 though there is no guarantee that the average perturbed normal is a remotely reasonable reconstruction of the true distribution of normals.

The single good thing about this approach is that there is an opportunity for sharing the same bump map encoding between the diffuse and specular illumination computations. The diffuse computation assumes an averaged perturbed normal that is *not* normalized while the specular computation requires the same normal normalized. Both needs are met by storing the normalized perturbed normal and a descaling factor. The specular computation uses the normalized normal directly while the diffuse computation uses the normalized normal but then fixes the diffuse illumination by multiplying by the descaling factor to get back to the unnormalized vector needed for computing the proper diffuse illumination. If $\mathbf{N'}_{\mathbf{filtered}}$ is computed according to Equation 14, then the normalized version is

$$\frac{\mathbf{N'}_{filtered}}{\left|\mathbf{N'}_{filtered}\right|}$$

and the descaling factor is simply the length of $\mathbf{N'}_{\mathbf{filtered}}$.

2.6 Bump Map Representations

So far, we have discussed three distinct representations of bump maps but have not been very concrete with how each bump map representation is stored as a texture map. The three representations are height fields, offset maps, and normal maps. Each representation corresponds to one of the three equations developed previously for computing a perturbed surface normal. Height fields are used with Equation 5. Offset maps are used with Equation 6. Normal maps are used with Equation 9.

2.6.1 Height Fields

This is the most straightforward representation and one that is fairly easy to author. A height field bump map corresponds to a one-component texture map discretely encoding the bivariate function $F(u,v)$ described in Section 2.1.

Painting a gray-scale image with a 2D-paint program is one way to author height fields. Rendering bumpy surface with geometry and then reading back the resultant Z values from a depth buffer is another means to generate height fields.

Equation 5 actually uses the partial derivatives of F rather than F directly. Finite differencing can approximate these partial derivatives. This bump mapping approach is called *texture embossing* and is further described in Section 3.1.

2.6.2 Vector Offset Maps

Instead of computing the partial derivatives at rendering time, an offset map is constructed by pre-computing the derivatives from a height field and then encoding the derivatives in a texture map.

This corresponds to encoding $\mathbf{D}(u,v)$ from Equation 7 into a texture. In general $\mathbf{D}$ is a 3D vector, but because $\mathbf{D}$ is orthogonal to $\mathbf{N}$, an offset map can be encoded as a two-component texture if we assume some way to orient $\mathbf{D}$ with respect to $\mathbf{N}$. Typically $\partial\mathbf{P}/\partial u$ is used to orient $\mathbf{D}$. When used to orient $\mathbf{D}$, the normalized version of $\partial\mathbf{P}/\partial u$ is called the *tangent vector* and here labeled $\mathbf{T}_n$.

Lighting results can be computed in an arbitrary 3D coordinate system as long as all the vector parameters involved are oriented with respect to the same single coordinate system. This freedom allows us to select the most convenient coordinate system for lighting. Tangent space is just such a local coordinate system. The orthonormal basis for the tangent space is the normalized unperturbed surface normal $\mathbf{N}_n$, the tangent vector $\mathbf{T}_n$ defined by normalizing $\partial\mathbf{P}/\partial u$, and the binormal $\mathbf{B}_n$ defined as $\mathbf{N}_n \times \mathbf{T}_n$. The orthonormal basis for a coordinate system is also sometimes called the reference frame.

As will be demonstrated, it is convenient to rewrite $\mathbf{D}$ in tangent space. Equation 7 can be simplified by writing it in terms of $\mathbf{N}_n$ and $\mathbf{B}_n$ so that

$$\mathbf{D} = \frac{\partial F}{\partial u}\left(\mathbf{N}_n \times \frac{\partial \mathbf{P}}{\partial v}\right) - \frac{\partial F}{\partial v}\left|\frac{\partial \mathbf{P}}{\partial u}\right|\mathbf{B}_n$$

Equation 21

As observed by Peercy [21], $\partial\mathbf{P}/\partial v$ is in the plane of the tangent and binormal by construction so $\partial\mathbf{P}/\partial v$ can be expressed as

$$\frac{\partial \mathbf{P}}{\partial v} = \left(\mathbf{T}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right)\mathbf{T}_n + \left(\mathbf{B}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right)\mathbf{B}_n$$

Therefore

$$\mathbf{N}_n \times \frac{\partial \mathbf{P}}{\partial v} = \left(\mathbf{T}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right)\mathbf{B}_n - \left(\mathbf{B}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right)\mathbf{T}_n$$

because $\mathbf{B}_n = \mathbf{N}_n \times \mathbf{T}_n$ and $-\mathbf{T}_n = \mathbf{N}_n \times \mathbf{B}_n$.

By substituting this result into Equation 21, $\mathbf{D}$ becomes

$$\mathbf{D} = \frac{\partial F}{\partial u}\left(\left(\mathbf{T}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right)\mathbf{B}_n - \left(\mathbf{B}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right)\mathbf{T}_n\right) - \frac{\partial F}{\partial v}\left|\frac{\partial \mathbf{P}}{\partial u}\right|\mathbf{B}_n$$

Equation 22

This expresses $\mathbf{D}$ in a form not explicitly dependent on $\mathbf{N}_n$. Because $\mathbf{D}$ is expressed in tangent space, we can assign a convenient orientation to $\mathbf{N}_n$, $\mathbf{T}_n$, and $\mathbf{B}_n$ such as $\mathbf{T}_n$=[1,0,0], $\mathbf{B}_n$=[0,1,0], and $\mathbf{N}_n$=[0,0,1]. This makes is it very simple to compute $\mathbf{N'}$ in tangent space:

$$\mathbf{N'} = \mathbf{N} + \mathbf{D} = [\mathbf{0,0},c] + \mathbf{D} = [a,b,c]$$

Equation 23

where

$$a = -\frac{\partial F}{\partial u}\left(\mathbf{B}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right)$$

$$b = \frac{\partial F}{\partial u}\left(\mathbf{T}_n \bullet \frac{\partial \mathbf{P}}{\partial v}\right) - \frac{\partial F}{\partial v}\left|\frac{\partial \mathbf{P}}{\partial u}\right|$$

$$c = \left|\frac{\partial \mathbf{P}}{\partial u} \times \frac{\partial \mathbf{P}}{\partial v}\right|$$

Equation 24

Instead of having to encode $\mathbf{D}$ as a 3D vector requiring a three-component texture, this results means that the values of a and b can be encoded in a two-component offset map. The value of c is independent of the height field F and therefore does not need to be stored in the bump map. Instead c must be supplied from the surface parameterization during rendering. Often however c is assumed constant over the surface as will be developed below.

This new form of $\mathbf{D}$ is still inconvenient because $\mathbf{D}$ is still dependent on the surface partial derivatives. This dependency means that the bump map texture is still effectively tied to the surface parameterization. This dependency means that, in general, the encoding of the bump map cannot be decoupled from the surface. This makes it difficult to author a single bump map texture that can be applied to multiple objects. In principle, this requires a unique bump map texture for each distinct bump-mapped surface even when the surfaces share the same height field. Texture memory is typically a limited resource so it is very desirable to break this dependency.

As suggested by Peercy [21], if we are willing to make some assumptions about the nature of the parameterization of the

surface **P** and work in tangent space, we can remove the dependency of **D** on the surface parameterization. Consider when the surface parameterization is locally that of a square patch. This implies that $\partial\mathbf{P}/\partial u$ and $\partial\mathbf{P}/\partial v$ are orthogonal. This means that $\partial\mathbf{P}/\partial u \bullet \partial\mathbf{P}/\partial v = \mathbf{T}_n \bullet \partial\mathbf{P}/\partial v = 0$ and that $\partial\mathbf{P}/\partial u$ and $\partial\mathbf{P}/\partial v$ are equal in magnitude, i.e. $|\partial\mathbf{P}/\partial u| = |\partial\mathbf{P}/\partial v|$. Fortunately, this assumption is acceptable for many important surfaces such as flat polygons, spheres, tori, and surfaces of revolution.

By making the square patch assumption, **D** can be further simplified. Based on the square patch assumption, Equation 22 becomes

$$\mathbf{D} = -\frac{\partial F}{\partial u} k \mathbf{T}_n - \frac{\partial F}{\partial v} k \mathbf{B}_n$$

where

$$k = \left|\frac{\partial \mathbf{P}}{\partial u}\right| = \left|\frac{\partial \mathbf{P}}{\partial v}\right|$$

and then Equation 24 can be rewritten as

$$a = -\frac{\partial F}{\partial u}\frac{1}{k}$$
$$b = -\frac{\partial F}{\partial v}\frac{1}{k}$$
$$c = 1$$

Equation 25

If offset maps are used this way for bump mapping, the problem remains that **N'** is not normalized. As described in Section 2.5.2, this is not an issue if the diffuse and specular contributions are computed using a cube map texture. However, cube map textures cannot be reasonably oriented in tangent space because tangent space varies over the surface so the tangent space **N'** must be transformed by a rotation into the cube map's coordinate system (typically, world space).

Because the normal component of **D** in tangent space is zero, a 2x3 matrix is sufficient to rotate **D**. Given an unperturbed normal vector oriented in the cube map's coordinate system, the perturbed normal is computed as

$$\mathbf{N'}_x = \mathbf{N}_x + a\,\mathbf{A}_{00} + b\,\mathbf{A}_{01}$$
$$\mathbf{N'}_y = \mathbf{N}_y + a\,\mathbf{A}_{10} + b\,\mathbf{A}_{11}$$
$$\mathbf{N'}_z = \mathbf{N}_z + a\,\mathbf{A}_{20} + b\,\mathbf{A}_{21}$$

Equation 26

Though the matrix **A** is constant in the case of a flat plane, **A** varies over a curved surface because tangent space on a curved surface varies over the surface. Ideally, the rotation matrix should be updated per-pixel and be scale-invariant so as not to change the length of **D**.

Based on the discussion so far, $\partial F/\partial u$ and $\partial F/\partial v$ are encoded directly within an offset map. These differentials are signed values so the texture filtering hardware must be capable of filtering signed values to support offset maps.

Additionally, the differentials may vary over a potentially unbounded range, particularly when very steep bump maps are encoded. In general, hardware texture formats typically have a limited range, perhaps negative one to positive one for signed textures. Given the existence of the matrix **A**, one way to cope with the potentially unbounded range of differentials is to compose the rotation in Equation 26 with a uniform scale that extends the effective range of the offset map. This combined transform can then be loaded into the matrix **A** described above. The scale required varies depending on the maximum differential stored in any given offset map.

All this suggests that a signed two-component offset map with limited range is reasonable to support bump mapping using offset maps. Yet to account for reasonable pre-filtering for diffuse illumination as described in Section 2.4.2, an additional unsigned component should also be provided to store the average perturbed normal length when pre-filtering an offset map. In a bump-mapping scheme using diffuse cube maps, this additional component should be modulated with the diffuse cube map result. Otherwise, distant or small diffuse bump-mapped surfaces will appear too bright.

2.6.3 Normal Perturbation Maps

Much of the mathematics used to develop the offset map also applies to normal perturbation maps, particularly the concept of tangent space. The main difference between the two approaches is that normal maps encode rotations instead of offsets. The primary advantage of using rotations is that then the perturbed normal remains normalized.

Normal maps can be represented in a number of different equivalent representations that all are simply different ways to encode rotations. Quaternions, 3x3 matrices, and Euler angles are all possibilities.

Consider representing rotations as 3x3 matrices. Then Equation 9 can be rewritten more concretely as

$$\begin{bmatrix}\mathbf{N'}_x & \mathbf{N'}_y & \mathbf{N'}_z\end{bmatrix} = \begin{bmatrix}\mathbf{N}_x & \mathbf{N}_y & \mathbf{N}_z\end{bmatrix}\begin{bmatrix}R_{00} & R_{01} & R_{02}\\ R_{10} & R_{11} & R_{12}\\ R_{20} & R_{21} & R_{22}\end{bmatrix}$$

But this entire matrix multiply can be replaced with a direct encoding of the perturbed vector if the computation is done in tangent space! When **N** equals [0,0,1], then the above matrix multiply simplifies to

$$\begin{bmatrix}\mathbf{N'}_x & \mathbf{N'}_y & \mathbf{N'}_z\end{bmatrix} = \begin{bmatrix}R_{02} & R_{12} & R_{22}\end{bmatrix}$$

This makes tangent space an extremely efficient coordinate system for bump mapping computations.

The same mathematics developed in the previous section for computing perturbed normals applies to normal maps except that normal maps must renormalize the perturbed normal. The perturbed normal in Equation 23 is not normalized, but renormalizing it results in

$$N'_n = \frac{[a,b,c]}{\sqrt{a^2+b^2+c^2}}$$

Equation 27

Perturbed normals can be encoded using either the surface-dependent [*a*,*b*,*c*] from Equation 24, or the surface-independent [*a*,*b*,*c*] from Equation 25 by making the square patch assumption.

This requires a signed three-component normal map. Each component of the normal map is guaranteed to be within the negative one to positive one range. To avoid hardware support for signed texture filtering, the components can be "range-compressed" into the zero to one range typical of conventional texture hardware before download as part of the normal map construction process. In this case, the range-compressed perturbed normal is encoded as

$$N'_{nc} = \frac{(a,b,c)}{2\sqrt{a^2+b^2+c^2}} + \frac{1}{2}$$

Equation 28

After filtering when rendering, each range-compressed normal is then expanded back to its true negative one to one range for use in the lighting model.

When pre-filtering the three-component vector portion of the pre-filtered normal map, each averaged vector should be renormalized. This renormalized vector is used for computing specular illumination.

In addition to the normalized three-component vector in the normal map, an additional fourth component can store the unsigned averaged normal length before renormalization. This is simply

$$N'_{len} = \sqrt{a_{avg}^2 + b_{avg}^2 + c_{avg}^2}$$

After the diffuse dot product is computed, the diffuse illumination is then modulated by $\mathbf{N'}_{len}$. This prevents heavily filtered diffuse surfaces from looking too bright.

When the range-compressed vector form is used, this four-component normal map can be stored exactly like an 8-bit per component conventional RGBA texture so existing texture fetching and filtering hardware needs no modification for this type of normal map.

3 PREVIOUS HARDWARE APPROACHES

The various hardware bump-mapping implementations described in the literature use different variations on the principles presented in Section 2.

3.1 Texture Embossing

Texture embossing [22][24] is one of the simplest and most limited formulations of bump mapping. The fact that the technique can be accelerated with minimal hardware support has made texture embossing the basis for more than one hardware vendor's claim to support bump mapping "in hardware."[1] Unfortunately, texture embossing has a number of significant limitations that limit the technique's practical application.

Texture embossing uses a height field representation as described in Section 2.6.1 for its bump map representation. The height field for the surface to be bump mapped is stored in a scalar texture. Texture embossing numerically determines the partial differentials necessary for bump mapping from the height field texture during rendering.

The bump-mapped diffuse lighting contribution requires computing **L•N'**. Texture embossing assumes that **L** and **N'** are oriented in tangent space (as developed in Section 2.6.2). Then **N'** can be expressed as in Equation 23. For simplicity and to eliminate Equation 23's dependency on the surface parameterization, texture embossing makes the square patch assumption. So by combining Equation 23 and Equation 25

$$\mathbf{N'} = \left[\left(-\frac{\partial F}{\partial u}\frac{1}{k} \right) \quad \left(-\frac{\partial F}{\partial v}\frac{1}{k} \right) \quad 1 \right]$$

Unfortunately, the **N'** above is not normalized, but if we assume that $\mathbf{N'}_x$ and $\mathbf{N'}_y$ are small, then **N'** will be almost normalized. We can assume that *k* is 1 if we scale the height field appropriately. By making these two assumptions, the diffuse computation becomes

$$\mathbf{L} \bullet \mathbf{N'} = \mathbf{L}_z - \mathbf{L}_x \frac{\partial F}{\partial u} - \mathbf{L}_y \frac{\partial F}{\partial v}$$

Equation 29

The two terms with partial differentials can be approximated numerically using finite differences from the height field. So

$$-\mathbf{L}_x \frac{\partial F}{\partial u} - \mathbf{L}_y \frac{\partial F}{\partial v} \cong F(u,v) - F(u + \mathbf{L}_x \Delta u, v + \mathbf{L}_y \Delta v)$$

Equation 30

Evaluating Equation 30 requires differencing two height field texture accesses. The height field texture resolution determines the Δu and Δv displacements. For example if the height field is a 128x128 texture, then Δu and Δv are both 1/128, enough to shift by exactly one texel in each direction. Scaling by the light direction's $\mathbf{L}_x$ and $\mathbf{L}_v$ components further reduces these displacements. Bilinear filtering is required because the technique relies on texture coordinate displacements less than the scale of a texel.

The result of Equation 30 is a slope and therefore is a signed quantity. Adding this signed quantity to $\mathbf{L}_z$ computes **L•N'**.

Some intuition for embossing can be gained by considering a few basic scenarios. Consider the case of a flat height field. In this case, **F** is constant. This means the finite difference of **F** is always zero. Therefore **N'** = [0, 0, 1]. In this case, $\mathbf{L \bullet N'} = \mathbf{L}_z$ by Equation 29.

[1] NVIDIA's claim that the RIVA TNT and TNT2 graphics chips support hardware bump mapping is based on single-pass hardware support for texture embossing.

In cases where the surface slopes towards the light, the finite difference in Equation 30 will be positive and **L•N'** will increase beyond $\mathbf{L}_z$. Likewise, in cases where the surface slopes away from the light, the finite difference will be negative and **L•N'** will decrease from $\mathbf{L}_z$.

One case that does not work correctly is when the light direction matches the unperturbed surface normal. In this case, **L** = [0, 0, 1]. Equation 30 is always zero in this case because $\mathbf{L}_x$ and $\mathbf{L}_v$ are zero making the displacement always zero. The value of **L•N'** is computed incorrectly as merely $\mathbf{L}_z$. This anomaly results from texture embossing's failure to normalize **N'**. In this situation, texture embossed surfaces look flat and too bright. In general, the failure to normalize **N'** exaggerates the diffuse lighting contribution.

The technique's two height field texture accesses are well suited for dual-texturing hardware. Extended texture environment functionality such as NVIDIA's `NV_texture_env_combine4` extension [17] can provide the signed math operations to evaluate Equation 29. When dual-texturing hardware is unavailable, texture embossing can be implemented as a multi-pass algorithm using either subtractive frame buffer blending or the accumulation buffer to perform the necessary signed math.

Ideally, the texture coordinate displacements and $\mathbf{L}_z$ should be computed per-pixel based on a normalized **L**. In practice, these parameters are computed per-vertex and then linearly interpolated per-pixel. When surfaces are tessellated coarsely relative to the surface's degree of curvature, linear interpolation will produce objectionable artifacts due to the denormalization of **L**. Typically, the CPU is responsible for computing the tangent space vectors and texture coordinate displacements required for texture embossing. The per-vertex math required for computing the texture coordinate displacements is however uniform enough that it can be off-loaded to specialized per-vertex transform hardware. The GeForce and Quadro GPUs provide such a texture coordinate generation mode through the `NV_texgen_emboss` extension [15] to support texture embossing.

Texture embossing has several other issues. The basic embossing algorithm does not account for bumped surfaces facing away from the light direction. This is the situation when $\mathbf{L}_z$ is negative. Unfortunately, the finite differencing result from Equation 30 may be positive in this case, yet most hardware clamps negative color values to zero so the positive results from Equation 30 cannot be cancelled by the negative $\mathbf{L}_z$. Instead, such self-shadowed regions of the surfaces appear incorrectly lit. One solution is to force the values of $\mathbf{L}_x$ and $\mathbf{L}_y$ to zero at vertices where $\mathbf{L}_z$ is negative so Equation 30 must be zero. This must be done with care to minimize transition artifacts.

Texture embossing does not interact well with mipmap filtering. The texture displacements are scaled based on the height field texture resolution. If Δu and Δv are 1/128 and 1/128 for the base level 128x128 mipmap and mipmap filtering uses the 32x32 mipmap level, these displacements will be too small for the finite differencing to approximate a meaningful differential. Bumpy surfaces rendered with texture embossing and mipmap filtering go flat when smaller level mipmaps are used. The problem is that texture embossing is tied to a single height field

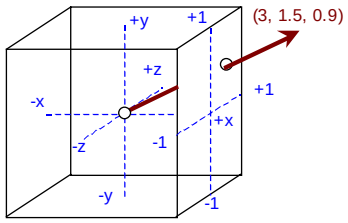

Figure 5. The unnormalized direction vector (3, 1.5, 0.9) pierces the positive X face of the cube map and projects to the 2D position (0.5, 0.3) on the face.

scale. Without mipmap filtering, the technique results in objectionable aliasing artifacts when mipmapping would otherwise be helpful.

While texture embossing is well suited to today's hardware with dual textures and extended texture environment functionality, the technique suffers from several significant issues. The technique is not robust and has failed to be adopted by applications beyond demos.

3.2 Dedicated Hardware Schemes

Graphics hardware researchers have proposed a variety of approaches for implementing bump mapping through dedicated hardware schemes. This section briefly reviews the literature on other hardware bump mapping approaches.

Evans and Sutherland and SGI have both proposed bump-mapping approaches for high-end 3D graphics hardware [5][21]. Both approaches propose to support bump mapping in the context of hardware that supports per-fragment lighting operations. The SGI technique relies on lighting in tangent space. The technique described in this paper borrows heavily from SGI's notion of tangent space.

Researchers at the German National Research Center for Information Technology have described several approaches for implementing hardware bump mapping. One of their approaches referred to as Visa+ bump-mapping [1][6] uses vector offset maps to encode the bump map perturbations and uses two cube maps to determine the respective lighting diffuse and Phong specular lighting contributions. This is the approach critiqued earlier in Section 2.5.2.

Another approach developed by the same researchers is called Gouraud bump mapping [7]. This approach bears some similarities to the technique described in this paper. Both techniques aim for a low-cost solution.

Microsoft includes a bump mapping mechanism in DirectX 6 called BumpEnv [19]. While the inference from the name is that the mechanism supports environment-mapped bump mapping, the hardware functionality is really a fairly limited dependent texture mechanism. Two sets of texture coordinates are supported. When BumpEnv is in use, a 2D perturbation texture is accessed using the first set of texture coordinates. This fetch returns a signed perturbation (*dx*,*dy*). Then this signed perturbation is added to the second set of texture coordinates. A 2x2 matrix then transforms the perturbed texture coordinates.

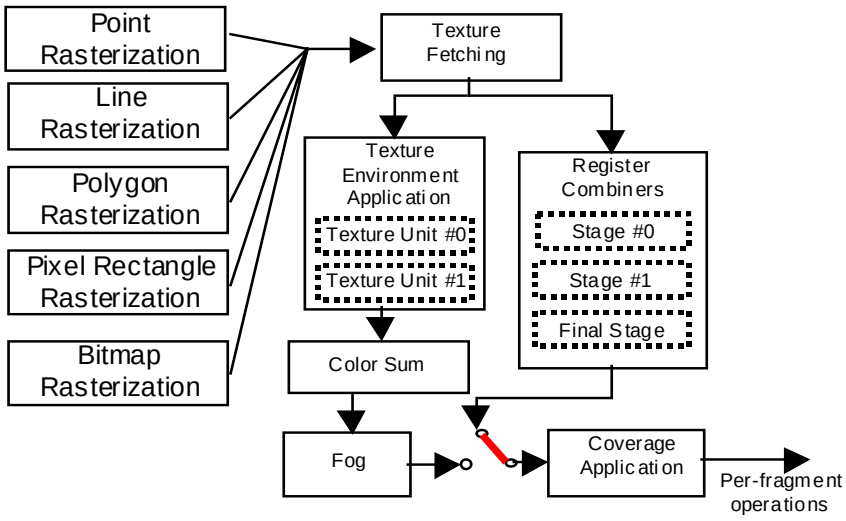

Figure 6. The data flow for register combiners and the standard OpenGL fragment coloring operations. The register combiners enable selects between the two fragment coloring pipelines. This figure is based on Figure 3.1 from the OpenGL 1.2.1 specification [25].

Lastly, the perturbed and transformed texture coordinates are used to access a second 2D texture containing color data.

The BumpEnv mechanism's claim to implement bump mapping is somewhat tenuous. If the approach supported a 2x3 matrix and accessed a cube map texture along the lines of Visa+ approach, the bump-mapping claim could be better justified. As it exists today, the BumpEnv functionality can be considered a degenerate form of Visa+ bump mapping with only a single cube map face instead of six faces of two fully populated cube maps. This is not to say that BumpEnv is not useful. Under limited conditions such as flat or relatively flat surfaces with relatively small perturbations illuminated by only directional lights (water outdoors for example), BumpEnv can arguably accomplish bump mapping. However, many of BumpEnv's robustness issues are direct results of the corners the approach cuts.

4 NVIDIA GPU FEATURES

The technique described in Section 5 heavily relies on two new features supported by both the GeForce and Quadro GPUs. This section introduces these new features. The first feature is support for cube map textures. The second feature is a greatly extended fragment coloring mechanism called the *register combiners*.

4.1 Cube Map Texturing

Cube map texturing is a form of texture mapping that uses an unnormalized 3D direction vector (s,t,r) to access a texture that consists of six square 2D images arranged like the faces of a cube centered around a coordinate system origin.

Figure 5 shows how an unnormalized direction vector [3, 1.5, 0.9] is mapped to a 2D location on the correct cube map face. First, the face that the vector pierces is determined. This is accomplished by determining which of the vector's components has the largest magnitude. The greatest magnitude component and its sign determine the pierced cube map face. There are six possibilities, each corresponding to one of the six cube faces: $+x$, $-x$, $+y$, $-y$, $+z$, and $-z$. In the case of [3, 1.5, 0.9], the largest magnitude is the x component's value 3.0. The component is positive so the $+x$ face is selected. Second, the remaining two components are divided by the component of the largest magnitude. This effectively projects these two components onto the selected cube map face. Finally, the two projected coordinates are scaled and biased depending on the selected face to compute a 2D texture coordinate used to access the texture image for the selected face. From this point on, the 2D texture coordinate is used to access the selected face's texture image just as if it was a standard 2D texture.

Cube map texturing hardware performs the face selection, projection, and scale and bias operations per-fragment to access cube map textures. The most expensive operation in determining the projected 2D texture coordinate on a particular cube map face because projection requires a divider. A similar per-fragment division is required for perspective correct 2D texturing. By reusing this single per-pixel divider for both 2D texturing and cube map texturing, the additional required hardware to support cube map texturing is nominal.

Both OpenGL and Direct3D support cube map texturing through standard API interfaces. Direct3D added cube map support with its DirectX 7 release. Direct3D and OpenGL share the same arrangement of cube faces with each other and Pixar's RenderMan API.

Cube maps are more flexible than other hardware-accelerated approaches for texturing based on direction vectors such as sphere maps and dual-paraboloid maps [12]. While sphere maps are view dependent, cube maps are view independent. While dual-paraboloid maps are view independent too, dual-paraboloid maps require two texture applications instead of the

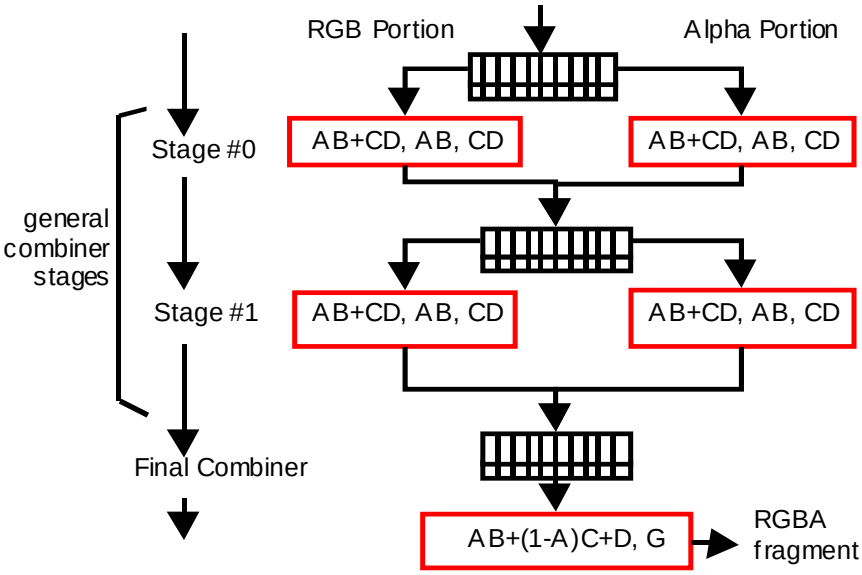

Figure 7. The data flow for register combiner stages.

single texture application required for cube maps. And both sphere maps and dual-paraboloid maps require a non-linear warping of a set of cube faces to construct the required textures. Cube maps can obviously use cube face images directly.

Cube map texturing is closely associated with environment mapping [2][11][27] because the cube face images can capture a complete omnidirectional view of the environment around a point. However, environment mapping is just one application of cube maps. The bump-mapping technique described in Section 5 uses cube maps to perform per-fragment normalization of interpolated light and half-angle vectors.

Further information about OpenGL programming of cube map textures is found in Appendix A.

4.2 Register Combiners

The GeForce and Quadro register combiners functionality provides a configurable (but not programmable) means to determine per-pixel fragment coloring. When enabled, the register combiners replace the standard OpenGL per-fragment texture environment, color sum, and fog operations with a greatly extended mechanism for coloring fragments. Figure 6 shows this data flow.

With multi-textured OpenGL, filtered texels from each texture unit are combined with the fragment's current color in sequential order. The color sum and fog stages immediately follow the texture environment in a fixed sequence. The standard fragment coloring mechanisms provided by both OpenGL and Direct3D are not particularly flexible and limit the scope of operations that can be performed on a given fragment.

The register combiners expose a sequence of general combiner stages that terminate in a final combiner stage that outputs an RGBA color for the fragment. The register combiners support the following functionality:

- Multiple combiner inputs are available in each combiner stage. All enabled textures, the primary (diffuse) and secondary (specular) colors, the fog color and factor, two constant colors, and two spare inputs are available.
- Computations in each general combiner stage use a signed numeric range from [-1,1] instead of an unsigned [0,1] range.
- The numeric range of each input is mapped and possibly clamped using one of eight distinct input mappings. These input mappings provide conversions from unsigned to signed numeric ranges, negation, half-biasing, and unsigned inversion.
- The RGB and alpha portions are configured and processed independently.
- Each general combiner stage outputs three distinct outputs for both the RGB and alpha portions.
- Possible outputs are products of inputs, a sum of products of inputs, 3-element vector dot products of RGB inputs, or a mux of products of inputs.
- Each stage writes its outputs to a set of registers that become the inputs for the subsequent stage. Unwritten register values carry forward from one stage to the next.
- A special final combiner stage combines the final register values into a final RGB and alpha result for each fragment.

4.2.1 The Register Set

When fragment coloring via the register combiners begins for a given fragment, a set of 4-element vector "registers" is initialized with RGBA parameters. Some of the registers contain varying parameters. These varying parameters are the interpolated primary (diffuse) and secondary (specular) colors, the filtered texels from enabled texture units, and the fog factor (though the fog factor is only available for use in the final combiner). These varying registers can be both written and read as part of combiner operation.

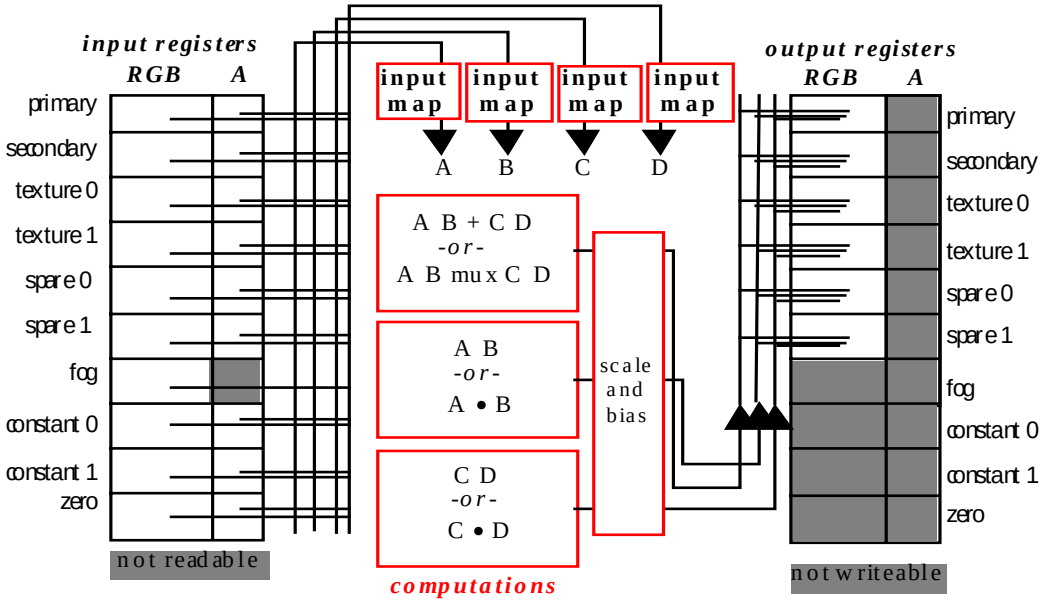

Figure 8. General combiner operation for the RGB portion of the combiner.

Another set of registers is uninitialized but can also be both written and read; these registers (called *spare0* and *spare1*) are intended as scratch registers. For reasons to be explained later, the alpha portion of the *spare0* register is actually initialized to the alpha component of texture unit 0 if texture unit 0 is enabled.

A final set of registers contains constants such as the RGB fog color, the value zero, and two RGBA color constants. These registers can be read by combiner operations but cannot be written by combiner operations. However, the fog color and the two constants can be specified by the application outside of rendering.

4.2.2 General Combiner Stage Operation

The GeForce and Quadro GPUs support two general combiner stages and a final combiner stage. Later NVIDIA GPUs will support more general combiner stages. There is always a single final combiner. Figure 7 shows how the general combiner stages proceed in sequence and terminate with the final combiner stage emitting the fragment's RGBA color.

All of the general combiner stages operate identically. Each general stage has an RGB portion and an alpha portion. The computations performed by the two portions of each stage are similar except that computations on the RGB portion operate on 3-element RGB vectors while computations on the alpha portion operate on scalar values. One important difference between the RGB and alpha portions is that the RGB portion also can compute 3-element dot products.

Figure 8 and Figure 9 show the operation of the RGB and alpha portions respectively of each general combiner.

Each portion of each general stage has four variables: A, B, C, and D. The value of each variable is assigned from one of the registers. The value for an RGB variable can be either the RGB portion of a register or the alpha portion of a register smeared into a vector. The value for an alpha variable can be either the alpha portion of the register or the blue component of the RGB portion of a register. Upon assignment, one of eight input mappings is performed on the input value. These input mappings can expand an unsigned value into a signed value, half-bias, negate, or perform an unsigned invert on the input value.

Once the values for A, B, C, and D are assigned for a given portion of a particular stage, three output values are computed based on the values of A, B, C, and D. These output values are scaled and biased and then clamped to the [-1,1] range and written back to distinct registers in the fragment's register set. The scale can be by one half, one, two, or four. The bias can be by either negative one half or zero. The scale and bias are the same for all outputs of a given portion of a general combiner stage (so the RGB and alpha portions can each have a distinct scale and bias). Each of the three outputs must be written to a distinct register. Optionally, the value of any output can be discarded without writing it to a register.

The first output value is either the product of A and B or the dot product of A and B. The second output value is either the product of C and D or the dot product of C and D. The product outputs are scalar operations for the alpha portion and vector operations for the RGB portion. The dot product outputs are only possible for the RGB portion of a general combiner stage (it only makes sense for vector data). The dot product result is a scalar so it is smeared across RGB for output as an RGB vector. The AB and CD product and dot product operations are signed computations.

The third output is either AB+CD (where AB and CD are products) or a mux (selection) between the AB and CD products. The AB+CD operation is performs a signed addition. The mux operation outputs the AB product if the alpha value of the first spare register (called *spare0*) is less than 0.5 and

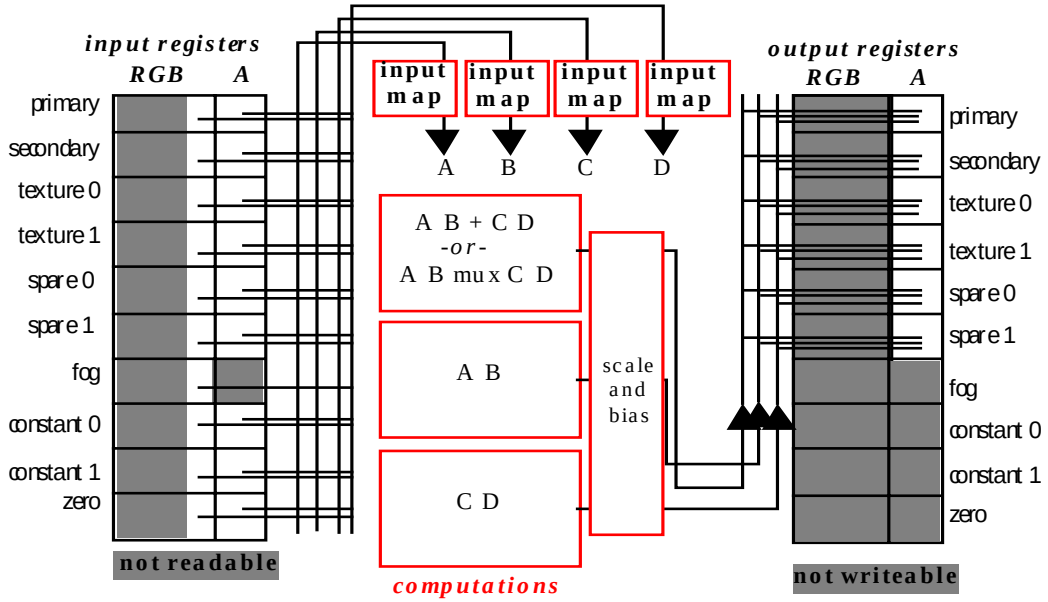

Figure 9. General combiner operation for the alpha portion of the combiner.

otherwise outputs the CD product. This use of *spare0* alpha value is why the *spare0* register's alpha value is initialized to the texture unit 0's alpha value if the texture unit is enabled.

Each portion of each general combiner stage can output up to three values. So the RGB and alpha portions of each general combiner stage can together output up to six values. With two general combiner stages, there is the opportunity to compute up to 12 per-fragment results with 4 possible dot products!

Once the outputs are written to the register set or discarded, the next general combiner in the sequence operates the same way. Any register values not written by a stage retain their values in the subsequent stage.

The input variable assignments, input mappings, component usage, output options, output registers, and output scale and bias can all be specified by the application for each portion of each general stage.

4.2.3 Final Combiner Stage Operation

After the general combiners complete their operations, the resulting register set is supplied to the final combiner to compute the fragment's final RGBA fragment color. Whereas the result of general combiner stages is simply the updating of the register set, the result of the final combiner is a single RGBA value. Figure 10 shows the operation of the final combiner.

The final combiner computes an unsigned result using unsigned inputs. Negative register values are clamped to zero before use in the final combiner. The final combiner has six variables used to compute the fragment's RGB color: A, B, C, D, E, and F. Each variable can input from any of the available registers. Each input can be inverted and can use either the RGB or alpha component of its input register. There are also two final combiner *pseudo-registers* that provide extra computations. The *EF product* pseudo-register can supply the input for the A, B, C, and D variables and evaluates to the product of the E and F variables. The *spare0+secondary color* pseudo-register can supply the B, C, and D variables and evaluates to the RGB addition of the *spare0* and secondary color registers.

The final combiner computes AB+(1-A)C+D to determine the fragment's RGB color. This is an RBG vector computation. The application can control whether the *spare0+secondary color* register clamps to 1.0 or not. When not clamped the *spare0+secondary color* register ranges from 0.0 to 2.0.

A seventh variable G directly determines the fragment's alpha value. This variable can input from the alpha portion of any register except the pseudo-registers. Optionally, the G input can be inverted.

4.2.4 Practical Details

The register combiners functionality is exposed to applications through the `NV_register_combiners` OpenGL extension. There is no direct access to the register combiners through Direct3D though an extension to Direct3D to expose the register combiner functionality has been proposed.

The same hardware resources implement the register combiners functionality and OpenGL and Direct3D's existing fragment coloring functionality. When using conventional OpenGL or Direct3D fragment coloring, the 3D driver converts the API's conventional fragment coloring operations into an equivalent register combiners configuration.

The number of active general combiners (1 or 2, possibly more on future GPUs) can be controlled by the application. While performance is hard to characterize completely and can change from implementation to implementation, the GeForce and Quadro GPUs can run at full pixel rate when a single general combiner is active, but the peak pixel rate drops in half when two general combiners are active. However, the GeForce and Quadro GPUs similarly run at half rate when two linear-mipmap-linear textures are enabled. This means that two gen-

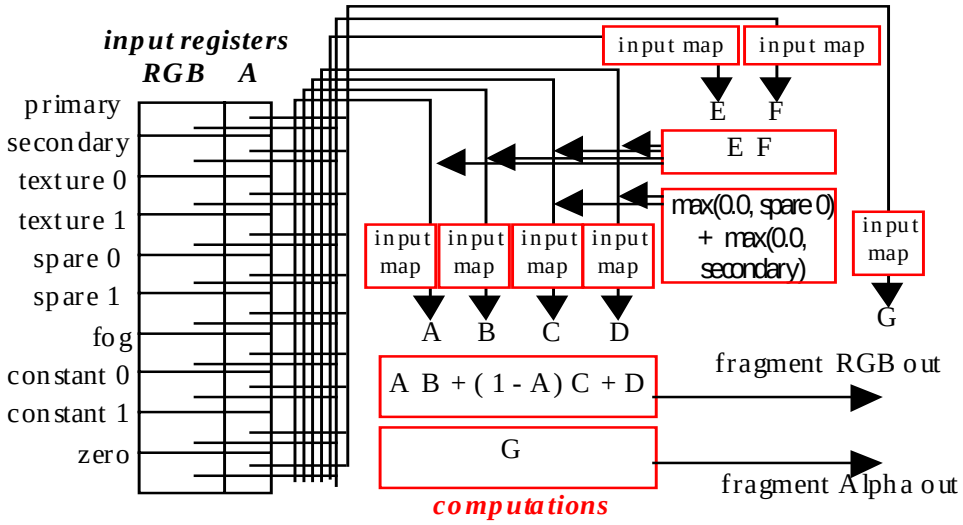

Figure 10. Final combiner operation.

eral combiners are "free" when two linear-mipmap-linear textures are active because the two enabled textures already cause the GPU to run at half rate. In practice, most cases when two general combiners are required are also cases where two textures are active.

Further information about OpenGL programming of register combiners is found in Appendix B.

5 A NEW BUMP-MAPPING TECHNIQUE

With the mathematics developed in Section 2 and the functionality presented in Section 4, we now describe the new bump-mapping technique. Where appropriate, practical C source code is presented in several appendices to make concrete the approach being described.

Before describing the details of the technique, here are several important observations about the overall approach:

- Bump maps are represented as normal perturbation maps. The pre-filtering scheme described in Section 2.6.3 is used to reduce aliasing artifacts, improve texture cache locality, and reasonably reproduce the roughness of distant bumpy diffuse surfaces.
- The per-fragment lighting operations employ the Blinn lighting model as described in Section 2.5.1. The required mathematics is built from the functionality available in the register combiners. The dot product operation is particularly useful for computing the diffuse and specular lighting terms.
- The technique uses multiple rendering passes. The technique typically renders three passes. The first pass renders a surface decal. The second pass modulates the decal with the bump-mapped diffuse and ambient contributions from a single light. The third pass adds a bump-mapped specular contribution from a single light. Both the second and third passes are dual-textured rendering passes.

  By expressing the technique as multiple rendering passes, the technique has a measure of scalability. For example, other lighting effects such as attenuation, spotlight illumination, and shadows can be added through further passes. Likewise, additional light sources can be supported with additional passes. The technique's rendering passes can also be scaled back when a decal texture is not required or when the object does not have a specular appearance.

  This multi-pass approach gives the technique an evolutionary path for future hardware acceleration. As more texture units and combiner stages are added in future GPUS, multiple rendering passes can be collapsed into fewer increasingly multi-textured passes.
- By lighting in tangent space as explained in Section 2.6.2, the technique allows the bump map texture to be decoupled from the object's geometric description.

  Tangent space is constructed on a per-vertex basis. This requires the CPU to supply tangent-space per-vertex light and half-angle vectors as 3D texture coordinates.

  The technique can also light in object space. This ties the bump maps to the object geometry, but this is often appropriate for surfaces such as terrain that have *ad hoc* variations in bumpiness over the surface. When lighting in object space, the CPU supplies object-space light and half-angle vectors as 3D texture coordinates.
- The light and half-angle vectors discussed above would become denormalized if they are simply linear interpolated. The technique performs per-fragment renormalizations of the interpolated light and half-angle vectors using a so-called "vector normalization" cube map.

Figure 11. Torus rendered with the new bump mapping technique. Specular, diffuse, and ambient lighting contributions are all present along with a surface decal texture.

This normalization cube map and the 2D normal map are the two textures used in the dual-textured second and third rendering passes.

Figure 11 shows the results of rendering a torus with the technique using three rendering passes. Figure 12 shows the contribution of each of the three passes.

5.1 Per-vertex Computations

Objects rendered with the technique must be represented as a polygonal model. As with any bump-mapping scheme, tangent vectors and normal vectors are required to orient the bump map consistently on the surface of the model. We assume that each vertex within the model has both a normal vector and tangent vector.

For polygonal models generated from parametric surfaces, these vectors are straightforward to generate from the parametric representation. Modeling packages typically supply per-vertex normals for use with standard per-vertex lighting, but tangent vectors are less common. If the modeling package generates per-vertex 2D texture coordinates for the object surface, tangent vectors can be deduced from these texture coordinates.

For tangent-space bump mapping, we make the square patch assumption as described in Section 2.6.2. If this assumption is not reasonable for your model, you should re-parameterize the model so the assumption is reasonable.

Given the per-vertex normal and tangent vectors for a polygonal model, we construct an orthonormal basis at each vertex. Using notation similar to the notation used in Section 2.6.2, we call the normalized normal vector $\mathbf{N}_n$ and the normalized tangent vector $\mathbf{T}_n$. The binormal $\mathbf{B}_n$ is defined as $\mathbf{N}_n \times \mathbf{T}_n$. These three normalized vectors form an orthonormal basis at each vertex.

We can then use this orthonormal basis to transform an object-space light and eye position into tangent space. As noted by Peercy [21], transforming an object-space vector into tangent space is done by transforming the vector by the 3x3 matrix formed by the three vectors $\mathbf{T}_n$, $\mathbf{N}_n$, and $\mathbf{B}_n$. So if $\mathbf{L}_{OS}$ is the object space light vector, the light vector in tangent space L_{TS} computed as

$$\mathbf{L}_{TS} = \mathbf{L}_{OS} \begin{bmatrix} \mathbf{T}_{n(x)} & \mathbf{B}_{n(x)} & \mathbf{N}_{n(x)} \\ \mathbf{T}_{n(y)} & \mathbf{B}_{n(y)} & \mathbf{N}_{n(y)} \\ \mathbf{T}_{n(z)} & \mathbf{B}_{n(z)} & \mathbf{N}_{n(z)} \end{bmatrix}$$

This same transform can be used to transform the object space half-angle vector from object space to tangent space as well.

Because most applications do not keep the eye and light positions in object space, applications may first need to transform these positions into object coordinates. This is simply a matter of transforming these positions in world space by the inverse matrix used to transform object vertices from object space to world space.

The tangent-space light and half-angle vectors must be recomputed whenever the light position, eye position, or the object's modeling transform change. For rigid models, the orthonormal basis for each vertex can be pre-computed once. Deformable models require updating the orthonormal basis of deformed vertices as well as re-computing the tangent-space light and half-angle vectors.

Both positional and directional lights can be handled by computing the object-space light vector appropriately. Supporting a local and infinite view model is likewise simply a matter of computing the object-space eye vector appropriately.

As an optimization, if the specular pass is not rendered, there is no need to compute tangent-space half-angles.

Appendix C presents source code that shows how to transform object-space light and eye vectors into tangent space and how to compute per-vertex tangent-space normalized light and half-

Figure 12. Torus rendered with the new bump mapping technique. Specular, diffuse, and ambient lighting contributions are all present along with a surface decal texture.

angle vectors. The code handles a positional light and a local viewer model. Note that this code is floating-point vector intensive and therefore could be substantially optimized using AMD's 3Dnow or Intel's SSE instructions.

5.2 Per-vertex Vector Interpolation and Normalization

During the ambient and diffuse bump-mapped rendering pass, the per-vertex tangent-space light vectors are sent to the hardware as (*s*,*t*,*r*) texture coordinates to access a "normalization" cube map texture to be described. Similarly during the specular bump-mapped rendering pass, the tangent-space half-angle is similarly sent to the hardware as (*s*,*t*,*r*) texture coordinates to access the same normalization cube map.

The vector normalization cube map texture contains RGB colors that when expanded in the register combiners are normalized per-fragment vectors. At this point the register combiners are configured to perform either diffuse or specular lighting calculations using either the normalized light or half-angle vector. The perturbed normal is supplied by the second texture unit bound to a 2D normal map texture.

For now, we consider the vector normalization cube map. In an abstract sense, cube maps can encode functions of unnormalized direction vectors. An environment map is really just a function that maps a reflected direction vector to a color. Based on this abstract conception of a cube map, note that vector normalization is a similar function. Instead of generating an RGB color from a given unnormalized direction vector, normalization merely generates a normalized version of the direction vector. Both an RGB color and a normalized vector are 3-element vectors. It should then be possible to encode the vector normalization function into a cube map.

One problem is that normalized vectors have values within the signed numeric range of [-1,1] while conventional texture formats support on the unsigned numeric range of [0,1]. However, by scaling by one-half and biasing by one-half, the [-1,1] range can be range-compressed into the [0,1] range. Getting back to the [-1,1] range requires reversing the original scale and bias by scaling by two and biasing by negative one. Conveniently, this range expansion is exactly what the `GL_EXPAND_NORMAL` register combiner input mapping does (see Table 3).

Note that linear texture filtering operations have the same final result whether the filtering is done in the [0,1] range and then expanded or in the true [-1,1] range. This means that the texturing hardware needs no special filtering allowances to store signed [-1,1] values compressed into the [0,1] range.

Constructing the range-compressed vector normalization cube map is straightforward. For each 2D texel position (*s*,*t*) on each face, compute the inverse of the cube map function that maps the 3D direction vectors (*s*,*t*,*r*) to a 2D (*s*,*t*) texel position on a given face image (see Section A.4).

This is an ambiguous inverse mapping because vectors in the same direction but of varying length all map to the same single 2D texel position on a face. Because we will normalize the result in any case, we can always simply multiply be one. The remaining coordinate is then either one or negative one depending on whether the face is the positive or negative face respectively for the axis. This 3D direction vector should then be normalized and range-compressed into the [0,1] range and treated as an RGB color. This is the color for the corresponding texel in the vector normalization cube map.

Appendix D presents source code that shows how to construct a normalization cube map for OpenGL.

5.3 Normal Map Construction

While the vector normalization texture is a cube map, the normal map is a more conventional 2D texture. Both textures however contain range-compressed vectors. In the case of the normal map, each texel encodes a perturbation vector used to perturb the object's surface to mimic surface bumpiness. This perturbation vector is stored in the RGB components of the texture, but the normal map is actually an RGBA texture as will be explained.

The base mipmap level of the normal map is likely to be generated from a height field. This is because height fields are convenient to author and store. Figure 13 shows the height fields used to generate the normal maps for the bump mapping examples in other figures.

A normal perturbation map is typically generated from a height field by applying Equation 25 and Equation 27 and using finite differencing techniques to compute the partial differentials. The resulting normalized perturbed normals must be range-compressed as shown by Equation 28 and stored in the RGB components of the normal map texture's base mipmap level.

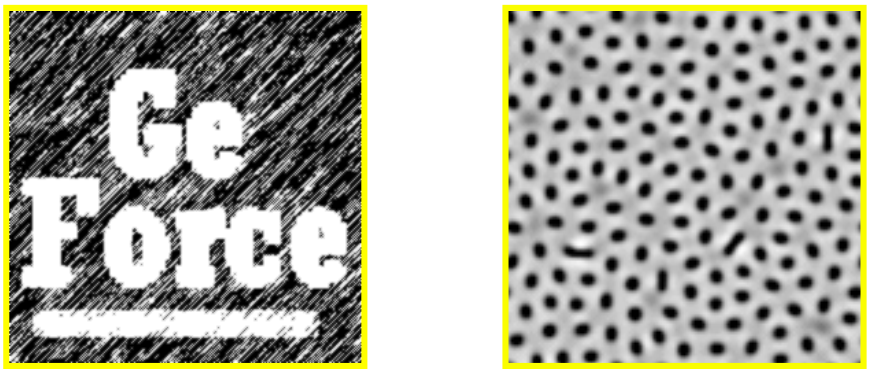

Figure 13. Two height field textures used to construct the normal maps used in the bump mapping examples within this paper.

The alpha value for the base level of the normal is uniformly 1.0.

The standard OpenGL `gluBuild2Dmipmaps` routine is not sufficient for mipmapping the normal maps. The normal map texture is filtered as described in Section 2.6.3. This means that the normal map also stores the amount of denormalization of the filtered normals in the texture's alpha component. While the base level alpha component of a normal map texture is uniformly 1.0, the alpha component of bumpy normal maps can shrink in the smaller mipmap levels and will typically be less than 1.0. The alpha component is not range-compressed. It is a signed quantity that can vary from 1.0 to 0.0.

The perturbation vector itself is always stored range-compressed and normalized in the RGB components of every mipmap level.

While better filtering is possible, the discussion below assumes a "box filter" approach to filtering the normal map. The mip-map filtering process for a normal map proceeds by down sampling each 2x2 block of texels in a given mipmap level to generate the corresponding single texel in the next smaller mipmap level. The filtering process is not a simple averaging of the 4 texels. Instead, first the RGB portion of each of the four texels is modulated by the texel's alpha component. This reverts the vector back to its denormalized length since the RGB components are always stored as a normalized vector. Then the denormalized vectors are averaged. The length of this averaged vector becomes the filtered texel's alpha component. The filtered texel's RGB components are the normalized and then range-compressed version of the averaged vector.

This filtering process is applied recursively to each mipmap level for the normal map.

This filtering approach gives access to both a normalized normal perturbation vector for specular lighting computations and a denormalized normal perturbation for diffuse lighting computations by modulating the normalized perturbation vector by the normal map's alpha component. This is advantageous for the reasons given in Section 2.4.2 and Section 2.5.3.

Appendix E presents source code that shows how to generate a normal map from a height field. Appendix F presents source code that shows how to load a normal map as an OpenGL texture including the proper filtering of mipmap levels.

5.4 Ambient and Diffuse Illumination

The first rendering pass is straightforward. If the object has a decal texture, enable depth buffering and render the object with the decal texture. Assuming the frame buffer supports an alpha component, a 1.0 alpha component should be written when rendering the first decal pass.

The second pass uses frame buffer blending to modulate the decaled object rendered in the first pass with the object's bump-mapped ambient and diffuse illumination from a light source.

Render the second pass with blending and depth testing enabled. The depth test should be set to equality to match the depth of fragments rendered in the first pass. The blend function should modulate the source and destination colors. OpenGL should be configured as

```
glEnable(GL_BLEND);
glBlendFunc(GL_DST_COLOR, GL_ZERO);
glDepthFunc(GL_EQUAL);
```

Optionally, stencil testing can be used to eliminate the possibility of double blending if that is a concern.

The bump-mapped ambient and diffuse illumination on the surface is computed with Equation 11 for every fragment belonging to the object. As will be shown, the register combiners can be configured to evaluate Equation 11.

The bump-mapped ambient and diffuse rendering pass enables both texture units. Texture unit 0 is bound to the 2D normal map texture. Texture unit 1 is bound to the vector normalization cube map.

While it introduces the possibility for slightly denormalizing the perturbed normal vectors, experience with the technique has proven that linear-mipmap-linear filtering of the normal map texture significantly reduces the temporal aliasing artifacts with minimal dimming of the overall appearance of the bump-mapped object.

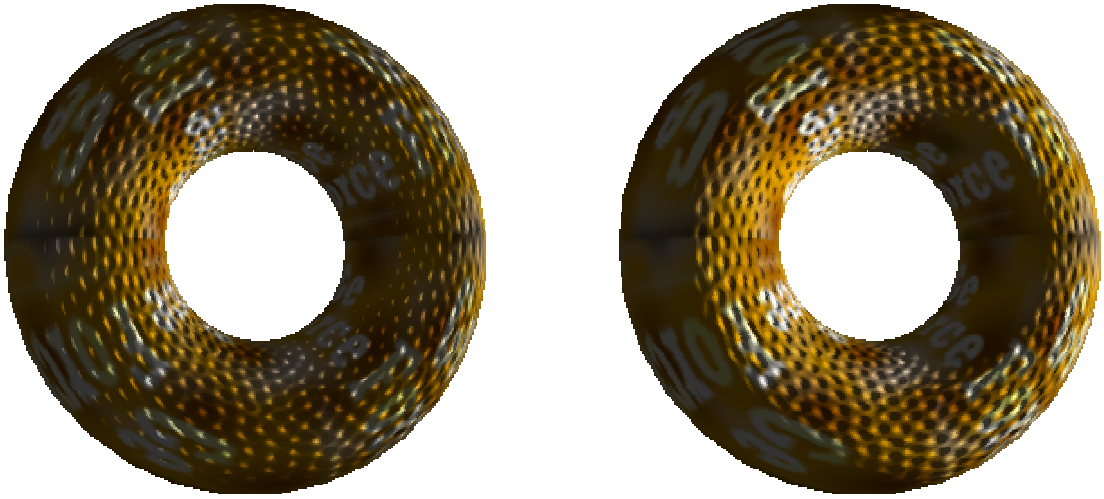

Figure 14. The left torus does not self-shadow based on the geometric normal. The right torus does self-shadow based on the geometric normal. The light position is the same for both tori, but the right torus gives a truer sense that the light is coming from the extreme right side.

Linear magnification filtering can give the surface a "grated" appearance due to denormalization when extreme magnification occurs. This can be substituted with "blockiness" by instead using nearest filtering. Another solution is to extrapolate the normal map's base level, being sure to range-expand, renormalize, and range-compress the texels in extrapolated mipmap levels. Extrapolation works by putting off when magnification occurs so it does not solve the problem. Additionally, extrapolation increases the normal map memory requirements.

The texture coordinates for the normal map are typically some version of the texture coordinates used to apply the decal in the first rendering pass.

Sometimes it is important for the decal to "match up" with the normal map. For example, the normal map might contain perturbations due to etched letters that must be aligned with letters visible in the decal. If the normal map was constructed from a height field aligned with the decal, the construction of the normal map by finite differencing as described in Section 5.3 and Appendix E will bias the normal map texture by half a texel in both *s* and *t*. The texture matrix can be used to compensate for this bias.

The texture coordinates for the vector normalization cube map are the per-vertex tangent-space light vectors described in Section 5.1.

Next, the register combiners must be configured to evaluate Equation 11 as follows:

1. Enable the register combiners and request two general combiner stages.
2. Load the ambient illumination $I_{ambient}$ in the constant color 0 register. Load the diffuse material characteristic $k_{diffuse}$ in the constant color 1 register.
3. In the RGB portion of general combiner stage 0, use the `GL_EXPAND_NORMAL` input mapping to assign and range-expand the RGB portions of the texture 0 and texture 1 registers into variables A and B. This loads the filtered normal perturbation in A and the normalized light vector in B. Output the dot product of A and B to the spare 0 register. This computes **L•N'** except that **N'** is normalized when it should be denormalized for proper filtering. Discard the other two outputs.
4. In the alpha portion of general combiner stage 0, use the ABCD output to scale the blue component of the light vector in texture 1 by 8. Use the `GL_EXPAND_NORMAL` input mapping to range-expand the blue component. The blue component of texture 1 is actually the *z* component of the tangent-space light vector, i.e. $\mathbf{L}_z$. Scaling $\mathbf{L}_z$ by 8 is accomplished by adding $\mathbf{L}_z$ to itself and scaling by 4. This computes the s_{self} self-shadowing term in Equation 11 as defined by Equation 12 once 8 times $\mathbf{L}_z$ is clamped to the range [0,1]. This is because $\mathbf{L}_z$ = **L•N** in tangent space. Output the value of 8 times $\mathbf{L}_z$ to the alpha portion of the spare 0 register. Discard the other two outputs.
5. In the RGB portion of general combiner stage 1, use the AB output to scale **L•N'** computed in stage 0 by the denormalization factor stored in the alpha portion of the texture 0 register. Load the RGB portion of the spare 0 register into A using the `GL_UNSIGNED_IDENTITY` input mapping to clamp **L•N'** to a positive value. Load the alpha portion of the texture 0 register into B. Output the product of A and B to the spare 0 register. Discard the other two outputs.
6. Discard all outputs of the alpha portion of general combiner stage 1.
7. In the final combiner, make the following assignments:
   - A equals the spare 0 alpha portion, i.e. the self-shadowing term s_{self}. The final combiner will automatically clamp s_{self} to a non-negative value.
   - B equals the EF product pseudo-register.

- C equals zero
- D equals the constant color 0, i.e. the ambient contribution $I_{ambient}$.
- E equals the spare 0 RGB portion, i.e. the diffuse term max (0,**L•N'**).
- F equals the constant color 1 RGB portion, i.e. the diffuse material reflectance $k_{diffuse}$.
- G equals spare 0 alpha portion, i.e. the self-shadowing term s_{self}. As with A, the value of s_{self} is automatically clamped to a non-negative value.

The resulting RGB color for the fragment is identical to evaluating Equation 11. The resulting alpha color for the fragment is the self-shadowing term s_{self}.

After blending, this leaves the ambient and diffuse illumination modulated with the object's decal. The frame buffer's alpha component contains the value s_{self}. This self-shadowing term must be modulated with the specular illumination term as well as the diffuse illumination. By stashing the s_{self} term in the alpha component of the frame buffer, the third specular rendering pass can modulate the specular illumination by a value computed by the second rendering pass using destination alpha frame buffer blending.

Figure 14 shows the difference that accounting for geometric self-shadowing makes by providing a strong cue to help the viewer determine where the light is located.

Appendix G presents source code that shows how to configure the register combiners to compute the ambient and diffuse illumination as described in this section.

5.5 Specular Illumination

A third rendering pass adds the specular contribution. This rendering pass configures the texture units as in the second pass. However, the blending mode, register combiners configuration, and texture coordinates for the normalization cube map texture are both quite different from the second pass.

The blend mode for the specular rendering pass adds the specular contribution to the existing diffuse and ambient contribution modulated with the surface decal. Yet as noted in the last section, the specular term should also be modulated with the self-shadowing term s_{self} stashed in the alpha component of the frame buffer during the second rendering pass. This requires the blend mode

```
glBlendFunc(GL_DST_ALPHA, GL_ONE);
```

Instead of supplying the tangent-space light vector for the normalization cube map texture coordinates, the tangent-space half-angle vector is supplied for the cube map's texture coordinates.

The register combiners are configured to compute the specular contribution from Equation 16 using a perturbed normal. The register combiners should be configured as follows:

1. Enable the register combiners and request two general combiner stages.
2. In the RGB portion of general combiner stage 0, use the `GL_EXPAND_NORMAL` input mapping to assign and range-expand the RGB portions of the texture 0 and texture 1 registers into variables A and B. This loads the normalized half-angle vector in A and the filtered normal perturbation in B. Output the dot product of A and B to the spare 0 register. This computes **H•N'**. Discard the other two outputs.
3. In the alpha portion of general combiner stage 0, use the `GL_UNSIGNED_IDENTITY_NV` input mapping to assign the blue component of texture 1 into variable A. This is $\mathbf{H}_z$ but is kept in the [0,1] range for reasons to be explained in Step 4. Assign 1 to variable B by assigning it zero input mapped using `GL_UNSIGNED_INVERT_NV`. Output the AB product to spare 0 alpha. This simply copies texture 1 alpha into spare 0 alpha. Discard the other two inputs.
4. In the RGB portion of general combiner stage 1, assign the RGB portion of spare 0 to both A and B using the GL_UNSIGNED_IDENITY_NV input mapping to clamp negative values in spare 0 to zero. The product of A and B is max(0, **H•N'**)^2. Assign C and D both zero. Use the mux output to output max(0, **H•N'**)^2 to spare 0 if spare 0 alpha is greater or equal to 0.5, otherwise output zero. This has the effect of clamping the specular contribution to zero if the unperturbed **H•N** is negative. Note that spare 0 alpha is greater than or equal to 0.5 when $\mathbf{H}_z$ is positive and $\mathbf{H}_z$ = **H•N**. Discard the other two outputs.
5. In the alpha portion of general combiner stage 1, discard all the outputs.
6. In the final combiner, make the following assignments:
   - A equals the EF product pseudo-register.
   - B equals the EF product pseudo-register.
   - C equals zero.
   - D equals zero.
   - E equals the RGB portion of spare 0
   - F equals the RGB portion of spare 0.
   - G equals zero.

The resulting RGB color for the fragment is max(0,**H•N'**)^8. The resulting alpha color for the fragment is zero. When this is modulated with s_{self} in the frame buffer's alpha component and added to the frame buffer's RGB components, this computes the total result of Equation 16 using **N'** the normal at each visible pixel in the bump-mapped object if we assume $K_{specular}$ is (1,1,1) and *shininess* is 8.

Unfortunately, 8.0 is not a particularly large specular exponent so the specular contribution is not particularly shiny. Also, the successive squaring of 8-bit precision values leads to banding artifacts though these artifacts are often hidden within the diffuse and ambient illumination. An alternative is to replace the specular exponentiation with a steep linear or quadratic ramp. For example, consider the approximation

$$\max(0, \mathbf{H} \bullet \mathbf{N'})^{shininess}$$

$$\cong$$

$$\max(0, 4 \times (\max(0, \mathbf{H} \bullet \mathbf{N'}) - 0.75))^2$$

Implementing the above approximation leaves a final combine multiplier free so the option exists to either square the above

approximation to make a still steeper ramp or use the multiply to modulate by $K_{specular}$.

Appendix H presents source code that shows how to configure the register combiners to compute the specular illumination as described in this section.

5.6 Other Effects

Other embellishments to the lighting model described so far are possible by rendering more passes. Attenuation can be added using either extra texturing passes or supplying a per-vertex fog coordinate based on distance to the light source using the `EXT_fog_coord` extension [18]. While not physically plausible, this scheme can support attenuation with exponential or exponential squared drop-offs in addition to more conventional inverse or inverse squared drop-offs.

Spotlights patterns can be encoded in projected textures as described by Segal [26].

Multiple light sources can be handled with multiple passes.

5.7 Without Special Features

While the discussion so far has assumed the availability of dual texturing, cube maps, and register combiners, it is possible to adapt aspects of the technique described to less functional graphics hardware.

Everitt has described a multi-pass technique called Orthogonal Illumination Mapping (OIM) [8] that computes dot products (or dot product approximations if subtractive blending is not supported) with multiple rendering passes. Unfortunately, an arbitrary dot product may require as many as 12 separate rendering passes. Various optimizations are possible such as using multitexture hardware to collapse certain passes into a single pass or exploiting the fact that one of the dot product operands is a constant vector.

And instead of cube maps, Heidrich's dual-paraboloid maps [12] can serve the same purpose that the vector normalization cube map serves. Again however, this approach would significantly increase the number of passes required, particularly if coupled with OIM.

Everitt [9] has demonstrated that tangent-space bump mapping along the lines of the techniques presented here is possible without resorting to special hardware features, but doing so required 26 rendering passes to approximate what a GeForce or Quadro GPU can do in 3 passes.

6 CONCLUSIONS

The described rendering technique implements a bump-mapped illumination model with ambient, diffuse, and specular contributions and a textured surface decal. The technique supports both positional or directional lights as well as either the infinite or local viewer models. The technique can light in either tangent space or object space though the presentation here has focused on the tangent space formulation.

The technique is robust. There are five notable reasons for the technique's robustness. First, the technique supports per-fragment normalization of interpolated light and half-angle vectors via the vector normalization cube map. This substantially reduces the need to tessellate the rendered polygon model as a fine mesh to avoid linear interpolation artifacts. Second, the normal map can encode completely arbitrary perturbations. The technique does not limit the range of possible perturbations. Third, by filtering the normal map properly and through the use of linear-mipmap-linear filtering, temporal aliasing artifacts when animating are minimal. The normal filtering scheme also faithfully reproduces the expected dimness of a distant bumpy object compared to a smooth version of the object. Fourth, the technique reasonably accounts of local surface self-shadowing effects. Fifth, the technique is not limited to directional lights so objects are free to move around and interact with the light source.

The technique is fast and works on mass-market 3D hardware that is widely available; the technique runs at quite interactive rates on today's GPUs. The extremely configurable register combiners mechanism makes it possible to implement the entire technique in a small, tractable number of rendering passes.

The ability to scale the technique to future hardware designs bears particular emphasis. Today the technique today requires three rendering passes for a single light source. However, it is not difficult to imagine extensions to future GPUs to support more texture units and more general combiner stages that would permit today's three-pass approach to require just a single rendering pass on a future GPU. Similarly, given future GPU enhancements and improved performance, it is easy to appreciate how embellishments such as spotlights, attenuation, shadows, and multiple light sources could be added through some combination of more rendering passes and/or more collapsing of existing multi-pass formulations into fewer multi-textured, multi-combiner passes.

Sample code for the technique discussed here is available from the Developer Relations section of the www.nvidia.com web site. Look for the `bumpdemo.zip` file in the OpenGL source code section.

REFERENCES

[1] K. Bennebroek, I. Ernst, H. Rüsseler, O. Wittig, "Design Principles of Hardware-based Phong Shading and Bump Mapping," *11th Eurographics Workshop on Graphics Hardware*, Poitiers, France, August 26-27, 1996, pp. 3-9.

[2] James Blinn, Martin Newell, "Texture and Reflection in Computer Generated Images," *Communications of the ACM*, 19(10), October 1976, pp. 542-546.

[3] James Blinn, "Models of Light Reflection for Computer Synthesized Pictures," *Computer Graphics (Proc. Siggraph '77)*, July 1977, pp. 192-198.

[4] James Blinn, "Simulation of Wrinkled Surfaces,'' *Computer Graphics (Proc. Siggraph '78)*, August 1978, pp. 286-292,. Also in *Tutorial: Computer Graphics: Image Synthesis*, pp. 307-313.

[5] Michael Cosman, Robert Grange, "CIG Scene Realism: The World Tomorrow," *Proc. Of I/ITSEC* on CD-ROM, 1996, pp. 628.

[6] I. Ernst, D. Jackèl, H. Rüsseler, O. Wittig, "Hardware Supported Bump Mapping: A Step towards Higher Quality Real-Time Rendering," *10th Eurographics*

Workshop on Graphics Hardware, Maastricht, Netherland, August 28-29, 1995, pp. 63-70.

[7] I. Ernst, H. Rüsseler, H. Schulz, O. Wittig, "Gouraud Bump Mapping," *Proc. 1998 Eurographics/Siggraph Workshop on Graphics Hardware*, Lisbon, Portugal, August 31-September 1, 1998, pp. 47-53.

[8] Cass Everitt, *Orthogonal Illumination Maps* web page, August 1999. http://www.opengl.org/News/Special/oim/Orth.html

[9] Cass Everitt, *Per-Pixel Lighting with Unextended OpenGL* web page, 1999. http://www.opengl.org/News/Special/oimupdate/per-pixel.html

[10] Alain Fournier, "Filtering Normal Maps and Creating Multiple Surfaces," University of British Columbia, Department of Computer Science, TR-92-41, 1992. http://www.cs.ubc.ca:80/cgi-bin/tr/1992/TR-92-41

[11] Ned Greene, "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications*, November 1986, pp. 21-30.

[12] Wolfgang Heidrich, Hans-Peter Seidel, "View-independent Environment Maps," *Proc. 1998 Eurographics/Siggraph Workshop on Graphics Hardware*, Lisbon, Portugal, August 31-September 1, 1998, pp. 39-45. http://www.mpi-sb.mpg.de/~heidrich/Papers/HWWS.1998.ps.gz

[13] NVIDIA Corporation, `EXT_texture_cube_map`, OpenGL extension specification, *NVIDIA OpenGL Extension Specifications*, September 1999. http://www.nvidia.com

[14] NVIDIA Corporation, `NV_register_combiners`, OpenGL extension specification, *NVIDIA OpenGL Extension Specifications*, September 1999. http://www.nvidia.com

[15] NVIDIA Corporation, `NV_texgen_emboss` OpenGL extension specification, *NVIDIA OpenGL Extension Specifications*, September 1999. http://www.nvidia.com

[16] NVIDIA Corporation, `NV_texgen_reflection` OpenGL extension specification, *NVIDIA OpenGL Extension Specifications*, September 1999. http://www.nvidia.com

[17] NVIDIA Corporation, `NV_texture_env_combine4` OpenGL extension specification, *NVIDIA OpenGL Extension Specifications*, September 1999. http://www.nvidia.com

[18] OpenGL Architectural Review Board, `EXT_fog_coord` OpenGL extension specification, included in the *NVIDIA OpenGL Extension Specifications*, September 1999. http://www.nvidia.com

[19] Kim Pallister, "Ups and Downs of Bump Mapping with DirectX 6," *Gamasutra* web site, June 4, 1999. http://www.gamasutra.com/features/19990604/bump_01.htm

[20] Bui Tuong Phong, "Illumination for Computer Generated Pictures," *Communications of the ACM*, 18(6), June 1975, pp. 311-317.

[21] Mark Peercy, John Airey, Brian Cabral, "Efficient Bump Mapping Hardware," *Computer Graphics (Proc. Siggraph '97)*, August 1997, pp. 303-306.

[22] Tom McReynolds, David Blythe, Brad Grantham, et al., "Bump Mapping with Textures," *Siggraph 1999 Course Notes: Lighting and Shading Techniques for Interactive Applications* or *Advanced Graphics Programming Techniques Using OpenGL*, Section 10.6, pp. 103-108.

[23] Andreas Schilling, "Toward Real-Time Photorealistic Rendering: Challenges and Solutions," Proc. 1997 *Eurographics/Siggraph Workshop on Graphics Hardware*, Los Angeles, California, August 3-4, 1997, pp. 7-15.

[24] John Schlag, "Fast Embossing Effects on Raster Image Data," *Graphics Gems IV*, Academic Press, Cambridge, 1994.

[25] Mark Segal, Kurt Akeley, *The OpenGL Graphics System: A Specification (Version 1.2.1)*, October 14, 1998.

[26] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, Paul Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping," *Computer Graphics (Proc. Siggraph '92)*, July 1992, pp. 249-252.

[27] Douglas Voorhies, Jim Foran, "Reflection Vector Shading Hardware," *Computer Graphics (Proc. Siggraph '94)*, July 1994, pp. 163-166.

A THE OPENGL CUBE MAP EXTENSION

This appendix is an introduction to programming OpenGL's cube map functionality. OpenGL supports cube map texturing via the `EXT_texture_cube_map` extension [13]. The OpenGL Architectural Review Board (ARB) has upgraded this extension to be an official ARB extension named `ARB_texture_cube_map`. The EXT and ARB extensions have identical semantics and share the same enumerant values so the only difference between the two extensions is the name.[2] The discussion below uses the EXT-suffixed enumerants and commands, but by merely changing the EXT suffix to ARB, the discussion applies identically to the ARB version of the extension.

A.1 New Cube Map Texture Targets

In conventional OpenGL, there are two types of textures: 1D and 2D textures. OpenGL 1.2 introduced a new 3D texture type for volumetric texturing. The texture cube map extension adds a new cube map texture type. OpenGL calls these different texture types *texture targets*. Including OpenGL 1.2 and the texture cube map extension, there are now four texture targets: 1D, 2D, 3D, and cube map. Each texture target has an associated enumerant that is passed to texture routines such as `glBindTexture`, `glTexParameter`, and `glTexImage`.

For example, the texture target for 2D textures is `GL_TEXTURE_2D`. This enumerant is passed as the first parameter to `glBindTexture`, `glTexParameter`, and `glTexImage` and similar calls. `glEnable` and `glDisable` also use the 2D texture target enumerant to enable and disable 2D texturing. Similarly the texture target enumerants for 1D and 3D textures are `GL_TEXTURE_1D` and `GL_TEXTURE_3D`.

For cube map textures, there is a new texture target enumerant called `GL_TEXTURE_CUBE_MAP_EXT`. The `EXT` suffix just indicates that the enumerant is for an `EXT` extension; all the new EXT cube map enumerants have this suffix. This is the enumerant that you should pass to `glBindTexture`, `glTexParameter`, `glEnable`, and `glDisable` when using cube map textures.

But the `GL_TEXTURE_CUBE_MAP_EXT` enumerant is *not* used for `glTexImage2D` and related commands such as `glCopySubTexImage2D`. The texture cube map extension makes a distinction not necessary for the other texture targets. While 1D, 2D, and 3D textures have only a single set of image mipmap levels, every cube map texture has six distinct sets of image mipmap levels. Each texture target can be mipmapped so the complete image data for 1D, 2D, and 3D textures is really a set of mipmap levels. For cube maps textures, each distinct cube map face has its own set of mipmap levels. Because cube maps have six faces (the other texture types can be thought of as having only one face each), the texture cube map extension makes a distinction between the cube map "texture as a whole" target and the six "texture image" targets. The six cube map texture image targets are:

```
GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT
GL_TEXTURE_CUBE_MAP_NEGATIVE_X_EXT
GL_TEXTURE_CUBE_MAP_POSITIVE_Y_EXT
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT
GL_TEXTURE_CUBE_MAP_POSITIVE_Z_EXT
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT
```

These targets correspond to the six cube map faces. If you think of a cube map texture as centered at the origin of and aligned to an *XYZ* coordinate system, each face is named by the positive or negative *X*, *Y*, or *Z* axis that pierces its face.

For convenience, the cube map texture image target enumerants are laid out in sequential order so that

```
GL_TEXTURE_CUBE_MAP_POSITIVE_Y_EXT = GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT + 2,
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT = GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT + 5 , etc.
```

These cube map texture image target enumerants are the target parameter values that should be used with `glTexImage2D`, `glCopyTexImage2D`, `glTexSubImage2D`, `glCopySubTexImage2D`, `glGetTexImage`, and `glGetTexLevelParameter` to update or query the specified image of the texture image target's respective cube map face.

Cube map images must always have square dimensions so the faces form a cube. In addition to the other texture consistency rules specified by OpenGL, all the faces at a given level of a cube map must have the same dimensions and the width and height of each particular image in the cube map must be equal.

Like the other texture targets, cube map textures support a special proxy target used to query if a given texture target configuration is supported by the OpenGL implementation. The cube map texture proxy target is `GL_PROXY_TEXTURE_CUBE_MAP_EXT`. You never use the proxy target for texturing. Instead you only query if it works or not. Because all the dimensions of all the faces for a given level of any cube map must have identical dimensions, there are not six proxy texture targets for each cube map face. A single cube map proxy target suffices.

---

[2] Early NVIDIA drivers for the GeForce and Quadro GPUs advertise the `EXT_texture_cube_map` extension and not the ARB extension because the ARB version of the extension had not been approved when the drivers were initially released. Subsequent NVIDIA drivers advertise both the EXT and ARB extensions for backward and future compatibility.

Major Axis Direction	Target	*sc*	*tc*	*ma*
$+rx$	GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT	$-rz$	$-ry$	rx
$-rx$	GL_TEXTURE_CUBE_MAP_NEGATIVE_X_EXT	$+rz$	$-ry$	rx
$+ry$	GL_TEXTURE_CUBE_MAP_POSITIVE_Y_EXT	$+rx$	$+rz$	ry
$-ry$	GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT	$+rx$	$-rz$	ry
$+rz$	GL_TEXTURE_CUBE_MAP_POSITIVE_Z_EXT	$+rx$	$-ry$	rz
$-rz$	GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT	$-rx$	$-ry$	rz

Table 1. Texture image face targets for each major axis direction.

For consistency with how OpenGL 1.2's 3D texture target is supported, there is also a new implementation defined constant GL_MAX_CUBE_MAP_TEXTURE_SIZE_EXT that indicates the maximum cube map texture size supported by the OpenGL implementation. In practice, the proxy mechanism is a preferable means to determine the implementation's specific limits.

A.2 Setting the Images for a Cube Map Texture

Here is how to load the six faces of a non-mipmapped cube map texture:

```
GLubyte face[6][64][64][3];
for (i=0; i<6; i++) {
  glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT + i,
    0,                  // level
    GL_RGB8,            // internal format
    64,                 // width
    64,                 // height
    0,                  // border
    GL_RGB,             // format
    GL_UNSIGNED_BYTE,   // type
    &face[i][0][0][0]); // pixel data
}
```

Each face in this example is a 64x64 RGB image.

Establishing mipmaps is not any more difficult. You can use the gluBuild2DMipmaps routine for establishing mipmap textures for cube map faces just like 2D textures. Instead of passing GL_TEXTURE_2D for the target parameter, pass in one of the "texture image" cube map targets. Example:

```
gluBuild2DMipmaps(GL_TEXTURE_CUBE_MAP_NEGATIVE_X_EXT,
  GL_RGB8, 64, 64, GL_RGB, GL_UNSIGNED_BYTE, &face[1][0][0][0]);
```

A.3 Enabling and Disabling Cube Map Textures

Enabling and disabling the cube map texture is done as follows:

```
glEnable(GL_TEXTURE_CUBE_MAP_EXT);
glDisable(GL_TEXTURE_CUBE_MAP_EXT);
```

As stated earlier, remember that for a cube map texture to be consistent, all the faces of all required levels must be square and have the same dimensions (in addition to the standard OpenGL texture consistency rules). If the texture is not consistent, OpenGL is required to act as if the inconsistent texture unit is disabled.

OpenGL has a priority scheme when multiple texture targets are enabled at the same time. Cube map texturing occurs when cube map texturing is enabled even if 3D, 2D, or 1D texturing is also enabled. The texturing enable priority is cube map, 3D, 2D, and finally 1D.

A.4 Mapping Texture Coordinates to Cube Map Faces

Because there are multiple faces, the mapping of texture coordinates to positions on cube map faces is more complicated than the 1D, 2D, and 3D texturing targets. The EXT_texture_cube_map extension is designed to be consistent with DirectX 7's cube map arrangement. This is also consistent with the cube map arrangement in Pixar's RenderMan package.

For cube map texturing, the (*s*,*t*,*r*) texture coordinates are treated as a direction vector (*rx*,*ry*,*rz*) emanating from the center of a cube. (The homogeneous *q* coordinate is ignored since it merely scales the vector without affecting the direction.) At texture application time, the interpolated per-fragment (*s*,*t*,*r*) selects one of the cube map face's 2D mipmap sets based on the largest magnitude coordinate direction (the major axis direction). The target column in Table 1 explains how the major axis direction maps to the 2D image of a particular cube map target.

Using the *sc*, *tc*, and *ma* determined by the major axis direction as specified in the table above, an updated (*s'*,*t'*) is calculated as follows

$$s' = \frac{sc}{2|ma|} + \frac{1}{2}$$
$$t' = \frac{tc}{2|ma|} + \frac{1}{2}$$

If $|ma|$ is zero or very nearly zero, the results of the above two equations may not be well defined (though the result may not lead to GL interruption or termination). Once the cube map face's 2D mipmap set and (s',t') is determined, texture fetching and filtering proceeds like standard OpenGL 2D texturing.

A.5 Texture Wrapping for Cube Map Textures

Ideally cube map texturing hardware would automatically filter texels from different faces when texture coordinates fell on or very near the edges and corners of the cube map. In practice, this is too complicated for current hardware. However, OpenGL 1.2's `GL_CLAMP_TO_EDGE` wrap mode[3] usually works sufficiently well that seaming artifacts from cube map face transitions are hardly noticeable.

The seaming artifacts are generally minor for one of two reasons. First, if the cube map texture is an environment map, humans are not particularly well adapted at finding faults in reflections off complex surfaces. Second, the cube map may encode a slowly changing function that makes seaming errors negligible. For example, the normalization cube map discussed in Section 5.3 changes very little across a seam. Moreover because the result of the normalization cube map feeds a subsequent lighting calculation, any seaming artifacts are not directly observable. Extreme minification or extremely low-resolution cube maps (as a rule of thumb, smaller than 32x32 texels on a face) are two cases where seaming artifacts are noticeable. In the case of extreme minification, such situations occur most often on distant objects or regions of high surface curvature making the artifacts difficult to notice.

For OpenGL implementations that support texture borders in conjunction with cube map textures,[4] one way to eliminate the seaming artifacts is to specify each face with a 1-texel border around each cube face that contains the data from the edge of the adjacent texture. At the corners of the border of each face, specify a weighted combination of the texels from the two other faces meeting at the corner. When mipmapping, these 1-texel borders can be added recursively to all the specified mipmap levels. When texture borders are added as described, then the `GL_CLAMP` wrap mode will eliminate any seaming artifacts.

Note that using the `GL_REPEAT` or `GL_CLAMP` with a constant border color is almost certain to cause objectionable seaming artifacts. While these modes are legal for cube map textures and should work on each face as specified for 2D texturing, the result in surely incorrect for cube maps.

The texture wrap parameters are specified using the "texture as a whole" cube map target. Once face selection is performed for a cube map texture, texture fetching and filtering operates just as it does for a 2D texture. This means that both the *s* and *t* wrap modes are used, but not the *r* wrap mode. Here is an example of setting the wrap mode for a cube map texture:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Note that because the default texture wrap mode is OpenGL is `GL_REPEAT`, you should always explicitly set the *s* and *t* wrap modes for cube maps textures.

A.6 Texture Coordinate Generation Modes for Cube Map Textures

You are free to pass per-vertex (*s*,*t*,*r*) texture coordinates for use with cube map texturing. For example:

```
glTexCoord3f(s,t,r);  /* user-supplied direction vector for cube map texturing */
glVertex3f(x,y,z);
```

In practice, it often makes sense to use one of OpenGL's texture coordinate generation modes. Two new texgen modes are added that generate the eye-space reflection vector or normal vector in the (*s*,*t*,*r*) texture coordinates. The reflection map mode can be enabled like this

```
glTexGenfv(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT);
glTexGenfv(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT);
```

---

[3] When OpenGL 1.2 is not supported, the `EXT_texture_clamp_to_edge` extension supplies the identical functionality.

[4] The GeForce and Quadro GPUs do not correctly support cube maps with texture borders; always use `GL_CLAMP_TO_EDGE` for cube maps on GeForce and Quadro.

Register Enumerant	Initial Value	Input/Output Status
GL_ZERO	Zero	Read-only
GL_CONSTANT_COLOR0_NV	Application-specified	Read-only
GL_CONSTANT_COLOR1_NV	Application-specified	Read-only
GL_FOG	Fog color and fog factor	Read-only
GL_PRIMARY_COLOR_NV	Interpolated primary (diffuse) color	Read/write
GL_SECONDARY_COLOR_NV	Interpolated secondary (specular) color	Read/write
GL_SPARE0_NV	RGB undefined, alpha is texture 0 alpha	Read/write
GL_SPARE1_NV	Undefined	Read/write
GL_TEXTURE0_ARB	Filtered texel for texture unit 0	Read/write
GL_TEXTURE1_ARB	Filtered texel for texture unit 1	Read/write
GL_TEXTURE*i*_ARB	Filtered texel for texture unit *i*	Read/write

Table 2. Register combiners register set.

```
glTexGenfv(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
```

The normal map mode can be enabled like this

```
glTexGenfv(GL_S, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_EXT);
glTexGenfv(GL_T, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_EXT);
glTexGenfv(GL_R, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_EXT);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
```

For these two modes to operate correctly, correct per-vertex normals must be supplied.

These new GL_REFLECTION_MAP_EXT and GL_NORMAL_MAP_EXT enumerants share the same respective values and functionality as the GL_REFLECTION_MAP_NV and GL_NORMAL_MAP_NV enumerants provided by the NV_texgen_reflection extension [16]. (The texgen reflection extension is typically used in the absence of cube maps to implement a dual-paraboloid map scheme [12]).

The GL_EYE_LINEAR texgen mode is also useful with cube maps as a way of generating the unnormalized view vector.

OpenGL's texture matrix is also very useful for manipulating cube map texture coordinates. The texture matrix can be used to rotate an (*s*,*t*,*r*) vector from one space to another. For example, consider if your cube map texture is oriented in world coordinate space where **M** is the matrix transform that moves from world coordinates to eye coordinates. You can load the inverse of the affine portion of **M** into the texture matrix to rotate the eye-space reflection or normal vectors generated by GL_REFLECTION_MAP_EXT or GL_NORMAL_MAP_EXT back into world space.

A.7 Cube Maps and Multitexture

Cube map texturing is orthogonal to multitexture. If the ARB_multitexture extension [25] is supported along with the EXT_texture_cube_map extension, all the texture units must support cube map texturing. This means you can mix 2D texturing and cube map texturing in a single pass if you have two texture units.

B THE OPENGL REGISTER COMBINERS EXTENSION

This appendix is an introduction to programming OpenGL's register combiners functionality. OpenGL supports register combiners via the NVIDIA-proprietary NV_register_combiners extension [14]. See Section 4.2 for an explanation of the register combiners data flow; this discussion concentrates on the registers combiners API.

B.1 Enabling and Disabling Register Combiners

Enabling and disabling the register combiners is done as follows:

```
glEnable(GL_REGISTER_COMBINERS_NV);
glDisable(GL_REGISTER_COMBINERS_NV);
```

Input Mapping Enumerant	Mapping Function	Range to Range
GL_UNSIGNED_IDENTITY_NV	max(0.0, x)	[0,1] ⇒ [0,1]
GL_UNSIGNED_INVERT_NV	1.0 – min(max(0.0, x), 1.0)	[0,1] ⇒ [1,0]
GL_EXPAND_NORMAL_NV	2.0 × max(0.0, x) – 1.0	[0,1] ⇒ [-1,1]
GL_EXPAND_NEGATE_NV	-2.0 × max(0.0, x) + 1.0	[0,1] ⇒ [-1,1]
GL_HALF_BIAS_NORMAL_NV	max(0.0, x) – 0.5	[0,1] ⇒ [-0.5,0.5]
GL_HALF_BIAS_NEGATE_NV	–max(0.0, x) + 0.5	[0,1] ⇒ [0.5,-0.5]
GL_SIGNED_IDENTITY_NV	x	[-1,1] ⇒ [-1,1]
GL_SIGNED_NEGATE_NV	–x	[-1,1] ⇒ [1,-1]

Table 3. Input mappings for variable inputs.

Parameter Enumerant	Initial Value	Type	Number of values
GL_CONSTANT_COLOR0_NV	(0, 0, 0, 0)	Color	4
GL_CONSTANT_COLOR1_NV	(0, 0, 0, 0)	Color	4
GL_NUM_GENERAL_COMBINERS_NV	1	Positive integer	1
GL_COLOR_SUM_CLAMP_NV	True	Boolean	1

Table 4. Register combiners parameters.

When disabled, fragments are colored by OpenGL's conventional texture environment application, color sum, and fog operations. When enabled, the current state of the register combiners is used to color fragments. Figure 6 shows this. Initially, the register combiners enable is disabled.

B.2 The Register Set

The number of registers in the register combiners register set is 8 plus the maximum number of texture units supported. Each register is referred to by its enumerant. Registers are either read/write or read-only unless otherwise noted. See Table 2. See Section 4.2.1 for more information about the register set.

B.3 Setting Combiner Parameters

Some register combiners parameters are not associated with a particular stage. These parameters are specified using the commands

```
glCombinerParameterfvNV(GLenum pname, const GLfloat *params);
glCombinerParameterivNV(GLenum pname, const GLint *params);
glCombinerParameterfNV(GLenum pname, GLfloat param);
glCombinerParameteriNV(GLenum pname, GLint param);
```

The four valid enumerants for *pname* for the above commands are listed in **Table 4** along with each parameter's initial value, type, and number of values.

B.4 Configuring General Combiner Stages

The `glCombinerInputNV` and `glCombinerOutputNV` commands configure the input and output state for each general combiner stage.

The `glCombinerInputNV` command specifies the input register, input mapping, and component usage for a particular variable (A, B, C, or D) for a particular general combiner stage and portion. The command has the following prototype:

```
glCombinerInputNV(GLenum stage,
                  GLenum portion,
                  GLenum variable,
                  GLenum input,
                  GLenum mapping,
                  GLenum componentUsage);
```

The *stage* parameter is an enumerant of the form `GL_COMBINER`*i*`_NV` where *i* is greater or equal to zero and less than the maximum number of general combiners supported. The maximum number of general combiners supported must be two or greater and can be determined by querying the `GL_MAX_COMBINERS_NV` implementation-defined limit. At least two general combiner stages are guaranteed.

The *portion* parameter must be either GL_RGB indicating the RGB portion of the combiner or GL_ALPHA indicating the alpha portion.

The *variable* parameter must be one of GL_VARIABLE_A_NV, GL_VARIABLE_B_NV, GL_VARIABLE_C_NV, or GL_VARIABLE_D_NV and determines which respective variable of the specified combiner stage and portion is updated.

The *input* parameter names a register from the list in Table 2. The *mapping* parameter names an input mapping from the list in Table 3. The *componentUsage* parameter must be either GL_RGB or GL_ALPHA if *portion* is GL_RGB or must be either GL_ALPHA or GL_BLUE if *portion* is GL_ALPHA. Also, if the *input* parameter is GL_FOG, the *componentUsage* may not be GL_ALPHA (the alpha portion of the fog register is the fog factor, which is not available for use until the final combiner stage).

Together, the *stage*, *portion*, and *variable* determine which variable's input state to update. The specified variable is assigned the value of the register named by *input* after the input mapping specified by *mapping* is performed. The portion of the register assigned is determined by *componentUsage*. If the *portion* is GL_RGB and *componentUsage* is GL_ALPHA, the register's alpha value is smeared to form an RGB vector.

The glCombinerOutputNV command specifies the output operations for a particular general combiner stage and portion. The command has the following prototype:

```
glCombinerOutputNV(GLenum stage,
                   GLenum portion,
                   GLenum abOutput,
                   GLenum cdOutput,
                   GLenum sumOutput,
                   GLenum scale,
                   GLenum bias,
                   GLboolean abDotProduct,
                   GLboolean cdDotProduct,
                   GLboolean muxSum);
```

The *stage* and *portion* parameters accept the same values accepted by glCombinerInputNV for these same parameter names. The *stage* and *portion* parameters determine the output state for a particular general combiner stage and portion that the remaining parameters will update.

The *abOutput*, *cdOutput*, and *sumOutput* parameters name registers (see Table 2) to which each particular output will be written. The register must be a read/write register. The parameter GL_DISCARD_NV is also accepted by these three parameters. If GL_DISCARD_NV is specified, the particular output is discarded. Each output register must be unique for the particular stage and portion specified; multiple outputs however can be discarded.

The *scale* parameter must be one of GL_NONE (meaning scale by 1.0), GL_SCALE_BY_TWO_NV, GL_SCALE_BY_FOUR_NV, or GL_SCALE_BY_ONE_HALF_NV. The *bias* parameter must be one of GL_NONE (meaning bias by 0.0) or GL_BIAS_BY_NEGATIVE_ONE_HALF_NV. The specified scale and bias is applied to all three outputs before they are written to their specified registers. If the *scale* parameter is either GL_SCALE_BY_ONE_HALF_NV or GL_SCALE_BY_FOUR_NV, then the *bias* parameter must be GL_NONE.

The *abDotProduct* parameter is zero if the AB output should compute a product and is non-zero if the AB product should instead compute a 3-element dot product. The *cdDotProduct* parameter is zero if the CD output should compute a product and is non-zero if the CD product should instead compute a 3-element dot product. The *abDotProduct* and *cdDotProduct* parameters must be zero if the *portion* parameter is GL_ALPHA. If either of the *abDotProduct* or *cdDotProduct* parameters are non-zero, then the *sumOutput* parameter must be set to GL_DISCARD_NV. The *muxSum* parameter is zero if the ABCD output should compute AB+CD and is non-zero if the ABCD output should mux (select) between the product AB or CD.

See Section 4.2.2 for more information on the operation of general combiner stages.

B.5 Configuring the Final Combiner Stage

The glFinalCombinerInputNV command specifies the input register, input mapping, and component usage for final combiner variables. The command has the following prototype:

```
glFinalCombinerInputNV(GLenum variable,
                       GLenum input,
                       GLenum mapping,
                       GLenum componentUsage);
```

Portion	Variable	Initial Input Register	Initial Component Usage	Initial Input Mapping
RGB	A	GL_PRIMARY_COLOR_NV	GL_RGB	GL_UNSIGNED_IDENTITY_NV
RGB	B	GL_TEXTURE#_ARB	GL_RGB	GL_UNSIGNED_IDENTITY_NV
RGB	C	GL_ZERO	GL_RGB	GL_UNSIGNED_IDENTITY_NV
RGB	D	GL_ZERO	GL_RGB	GL_UNSIGNED_IDENTITY_NV
alpha	A	GL_PRIMARY_COLOR_NV	GL_ALPHA	GL_UNSIGNED_IDENTITY_NV
alpha	B	GL_TEXTURE#_ARB	GL_ALPHA	GL_UNSIGNED_IDENTITY_NV
alpha	C	GL_ZERO	GL_ALPHA	GL_UNSIGNED_IDENTITY_NV
alpha	D	GL_ZERO	GL_ALPHA	GL_UNSIGNED_IDENTITY_NV

Table 5. Initial general combiner state for the both RGB and alpha portions. The # in GL_TEXTURE#_ARB is the number of the particular general combiner stage.

Final Stage Variable	Initial Input Register	Initial Component Usage	Initial Input Mapping
A	GL_FOG	GL_ALPHA	GL_UNSIGNED_IDENTITY_NV
B	GL_SPARE0_PLUS_SECONDARY_COLOR_NV	GL_RGB	GL_UNSIGNED_IDENTITY_NV
C	GL_FOG	GL_RGB	GL_UNSIGNED_IDENTITY_NV
D	GL_ZERO	GL_RGB	GL_UNSIGNED_IDENTITY_NV
E	GL_ZERO	GL_RGB	GL_UNSIGNED_IDENTITY_NV
F	GL_ZERO	GL_RGB	GL_UNSIGNED_IDENTITY_NV
G	GL_SPARE0_NV	GL_ALPHA	GL_UNSIGNED_IDENTITY_NV

Table 6. Initial final combiner state.

The *variable* parameter must be one of GL_VARIABLE_A_NV, GL_VARIABLE_B_NV, GL_VARIABLE_C_NV, GL_VARIABLE_D_NV, GL_VARIABLE_E_NV, GL_VARIABLE_F_NV, or GL_VARIABLE_G_NV, and these enumerants correspond to the variables A, B, C, D, E, F, and G respectively.

The *input* parameter names a register from the list in Table 2. The *input* parameter may also be either GL_E_TIMES_F_NV or GL_SPARE0_PLUS_SECONDARY_COLOR_NV; these additional two inputs correspond to the *EF product* and *spare0+secondaryColor* pseudo-registers. However, the EF product pseudo-register may only be used as an input for the A, B, C, and D variables. The spare0+secondaryColor pseudo-register may only be used as an input for the B, C, and D variables.

The *mapping* parameter must be either GL_UNSIGNED_IDENTITY_NV or GL_UNSIGNED_INVERT_NV. These input mappings operate as specified in Table 3.

The *componentUsage* parameter may be either GL_RGB or GL_ALPHA. However, the *componentUsage* must be GL_ALPHA if the *variable* parameter is GL_VARIABLE_G_NV. The *componentUsage* must GL_RGB for any variable inputting from either the EF product or spare0+secondaryColor pseudo-registers.

See Section 4.2.3 for more information on the operation of the final combiner stage.

B.6 Initial Register Combiners State

The register combiners enable is initially disabled.

The initial state for the inputs of each general combiner stage and portion is in Table 5.

The initial state for the outputs for each general combiner stage and portion is GL_NONE for the scale and bias, GL_DISCARD_NV for the AB output and the CD output, GL_SPARE0_NV for the ABCD output, and false for the AB dot product, the CD dot product, and the ABCD mux.

The initial state for the final combiner inputs is in Table 5.

The remaining initial register combiners state is listed in **Table 4**.

B.7 Combiner Queries

All the register combiners state can be queried. The following query commands are provided:

```
void glGetCombinerInputParameterfvNV(GLenum stage, GLenum portion, GLenum variable,
                                     GLenum pname, const GLfloat *params);
void glGetCombinerInputParameterivNV(GLenum stage, GLenum portion, GLenum variable,
                                     GLenum pname, const GLint *params);
void glGetCombinerOutputParameterfvNV(GLenum stage, GLenum portion,
                                      GLenum pname, const GLfloat *params);
void glGetCombinerOutputParameterivNV(GLenum stage, GLenum portion,
                                      GLenum pname, GLint *params);
void glGetFinalCombinerInputParameterfvNV(GLenum variable,
                                          GLenum pname, const GLfloat *params);
void glGetFinalCombinerInputParameterivNV(GLenum variable,
                                          GLenum pname, const GLfloat *params);
```

The `glGetCombinerInputParameterfvNV`, `glGetCombinerInputParameterivNV`, `glGetCombinerOutputParameterfvNV`, and `glGetCombinerOutputParameterivNV` parameter *stage* may be one of `GL_COMBINER0_NV`, `GL_COMBINER1_NV`, and so on, indicating which general combiner stage to query. The `glGetCombinerInputParameterfvNV`, `glGetCombinerInputParameterivNV`, `glGetCombinerOutputParameterfvNV`, and `glGetCombinerOutputParameterivNV` parameter *portion* may be either `GL_RGB` or `GL_ALPHA`, indicating which portion of the general combiner stage to query. The `glGetCombinerInputParameterfvNV` and `glGetCombinerInputParameterivNV` parameter *variable* may be one of `GL_VARIABLE_A_NV`, `GL_VARIABLE_B_NV`, `GL_VARIABLE_C_NV`, or `GL_VARIABLE_D_NV`, indicating which variable of the general combiner stage to query. The `glGetFinalCombinerInputParameterfvNV` and `glGetFinalCombinerInputParameterivNV` parameter *variable* may be one of `GL_VARIABLE_A_NV`, `GL_VARIABLE_B_NV`, `GL_VARIABLE_C_NV`, `GL_VARIABLE_D_NV`, `GL_VARIABLE_E_NV`, `GL_VARIABLE_F_NV`, or `GL_VARIABLE_G_NV`.

C COMPUTING TANGENT-SPACE LIGHT AND HALF-ANGLE VECTORS

This appendix shows how to compute normalized tangent-space light and half-angle vectors. The `updateTangentSpaceVectors` routine is passed an array of *numVertices* vertices and a second array of *numVertices* orthogonal basises for each corresponding vertex. The tangent-space light vector and half angle in each element of the *vertex* array is updated based on the vertex's corresponding orthonormal basis in the array *axis* and the object-space light and eye position referenced by *lightPosition* and *eyePosition* respectively.

```
typedef struct {                        /* A position or direction vector. */
  GLfloat x;
  GLfloat y;
  GLfloat z;
} Vector;

typedef struct {                        /* Vertex information. */
  Vector position;                      /* Object-space vertex position. */
  Vector tangentSpaceLightVector;       /* Tangent-space light vector. */
  Vector tangentSpaceHalfAngleVector;   /* Tangent-space half-angle vector. */
  GLshort s, t;                         /* Decal texture coordinates. */
} Vertex;

typedef struct {
  Vector tangent;                       /* Object-space tangent vector. */
  Vector binormal;                      /* Object-space binormal vector. */
  Vector normal;                        /* Object-space normal vector. */
} OrthoNormalBasis;

void updateTangentSpaceVectors(int numVertices, Vertex *vertex, OrthoNormalBasis *axis,
                               Vector *lightPosition, Vector *eyePosition)
{
  int i;
  float x, y, z, xx, yy, zz, invlen;

  for (i=0; i<numVertices; i++) {
    /* Subtract vertex position from the light position to compute the
       (unnormalized) direction vector to the light in object space. */
    x = lightPosition->x - vertex[i].position.x;
    y = lightPosition->y - vertex[i].position.y;
    z = lightPosition->z - vertex[i].position.z;
```

```
      /* Rotate the direction vector to the light into the vertex's
         tangent space. */
      xx = x*axis[i].tangent.x  + y*axis[i].tangent.y  + z*axis[i].tangent.z;
      yy = x*axis[i].binormal.x + y*axis[i].binormal.y + z*axis[i].binormal.z;
      zz = x*axis[i].normal.x   + y*axis[i].normal.y   + z*axis[i].normal.z;

      /* Normalize the tangent-space light vector. */
      invlen = 1.0/sqrt(xx*xx+yy*yy+zz*zz);
      vertex[i].tangentSpaceLightVector.x = xx*invlen;
      vertex[i].tangentSpaceLightVector.y = yy*invlen;
      vertex[i].tangentSpaceLightVector.z = zz*invlen;

      /* Subtract vertex position from the eye position to compute the
         (unnormalized) direction vector to the eye in object space. */
      x = eyePosition->x - vertex[i].position.x;
      y = eyePosition->y - vertex[i].position.y;
      z = eyePosition->z - vertex[i].position.z;

      /* Rotate the direction vector to the eye into the vertex's
         tangent space. */
      xx = x*axis[i].tangent.x  + y*axis[i].tangent.y  + z*axis[i].tangent.z;
      yy = x*axis[i].binormal.x + y*axis[i].binormal.y + z*axis[i].binormal.z;
      zz = x*axis[i].normal.x   + y*axis[i].normal.y   + z*axis[i].normal.z;

      /* Normalize the tangent-space eye vector. */
      invlen = 1.0/sqrt(xx*xx+yy*yy+zz*zz);
      xx = xx*invlen;
      yy = yy*invlen;
      zz = zz*invlen;

      /* Form the half-angle vector by adding the normalized light and eye vectors. */
      xx += vertex[i].tangentSpaceLightVector.x;
      yy += vertex[i].tangentSpaceLightVector.y;
      zz += vertex[i].tangentSpaceLightVector.z;
      invlen = 1.0/sqrt(xx*xx+yy*yy+zz*zz);

      /* Store the normalized tangent-space half-angle vector. */
      vertex[i].tangentSpaceHalfAngleVector.x = xx*invlen;
      vertex[i].tangentSpaceHalfAngleVector.y = yy*invlen;
      vertex[i].tangentSpaceHalfAngleVector.z = zz*invlen;
    }
  }
```

D CONSTRUCTING A NORMALIZATION CUBE MAP

This appendix shows how to construct a normalization cube map for use with the OpenGL cube map texture extension. The constructed cube map take an unnormalized 3D direction vector as an (*s*,*t*,*r*) texture coordinate and return an RGB vector that when expanded from the [0,1] range to the [-1,1] range using the register combiners' `GL_EXPAND_NORMAL` input mapping is the normalization of (*s*,*t*,*r*).

The `makeNormalizeVectorCubeMap` routine makes a cube map texture with a face resolution of *size* by *size* texels. The `getCubeVector` routine is a helper routine.

```
  static void getCubeVector(int i, int cubesize, int x, int y, float *vector)
  {
    float s, t, sc, tc, mag;

    s = ((float)x + 0.5) / (float)cubesize;
    t = ((float)y + 0.5) / (float)cubesize;
    sc = s*2.0 - 1.0;
    tc = t*2.0 - 1.0;

    switch (i) {  /* See Table 1 for the rationale for these cases. */
    case 0: vector[0] = 1.0;  vector[1] = -tc;  vector[2] = -sc;  break;
    case 1: vector[0] = -1.0; vector[1] = -tc;  vector[2] = sc;   break;
    case 2: vector[0] = sc;   vector[1] = 1.0;  vector[2] = tc;   break;
    case 3: vector[0] = sc;   vector[1] = -1.0; vector[2] = -tc;  break;
    case 4: vector[0] = sc;   vector[1] = -tc;  vector[2] = 1.0;  break;
```

```
    case 5: vector[0] = -sc;  vector[1] = -tc;  vector[2] = -1.0; break;
    }

    mag = 1.0/sqrt(vector[0]*vector[0] + vector[1]*vector[1] + vector[2]*vector[2]);
    vector[0] *= mag;
    vector[1] *= mag;
    vector[2] *= mag;
  }

  int makeNormalizeVectorCubeMap(int size)
  {
    float vector[3];
    int i, x, y;
    GLubyte *pixels;

    pixels = (GLubyte*) malloc(size*size*3);
    if (pixels == NULL) {
      return 0;  /* Memory allocation failed. */
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP_EXT, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    for (i = 0; i < 6; i++) {
      for (y = 0; y < size; y++) {
        for (x = 0; x < size; x++) {
          getCubeVector(i, size, x, y, vector);
          pixels[3*(y*size+x) + 0] = 128 + 127*vector[0];
          pixels[3*(y*size+x) + 1] = 128 + 127*vector[1];
          pixels[3*(y*size+x) + 2] = 128 + 127*vector[2];
        }
      }
      glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT+i, 0, GL_RGB8,
        size, size, 0, GL_RGB, GL_UNSIGNED_BYTE, pixels);
    }
    free(pixels);
    return 1;  /* Success. */
  }
```

E CONSTRUCTING A NORMAL MAP FROM A HEIGHT FIELD

This appendix shows how construct a normal map suitable for the described bump mapping technique. The routine `convertHeightFieldToNormal` map takes an array of *pixels* of size *wr* by *hr*. The routine generates an array of BGRA data of size *w* by *h*. If *wr* and *hr* are equal to *w* and *h* respectively, then the height field is assumed to wrap. If *wr* and *hr* are each one greater than *w* and *h* respectively, the *pixels* height field array is assumed to have an extra border of elements to the bottom and right that will be sampled (instead of wrapping) when constructing the height field. The returned array of data is intended to specify the base level of a 2D texture that will be used as a normal map. Because the returned array is intended for use as a texture image, the *w* and *h* parameters should be powers of two. Note that the color component ordering for this texture is **BGRA** since that is most efficient for texture download on NVIDA GPUs.

```
  /* Structure to encode a normal like an 8-bit unsigned BGRA vector. */
  typedef struct {
    /* Normalized tangent-space perturbed surface normal.  The
       [0,1] range of (nx,ny,nz) gets expanded to the [-1,1]
       range in the combiners.  The (nx,ny,nz) is always a
       normalized vector. */
    GLubyte nz, ny, nx;

    /* A scaling factor for the normal.  Mipmap level 0 has a constant
       magnitude of 1.0, but down-sampled mipmap levels keep track of
       the unnormalized vector sum length.  For diffuse per-pixel
       lighting, it is preferable to make N' be the _unnormalized_
       vector, but for specular lighting to work reasonably, the
       normal vector should be normalized.  In the diffuse case, we
       can multiply by the "mag" to get the possibly shortened
```

```
      unnormalized length. */
  GLubyte mag;
} Normal;

/* Convert a height field image into a normal map.  This involves
   differencing each texel with its right and upper neighbor, then
   normalizing the cross product of the two difference vectors. */
Normal *convertHeightFieldToNormalMap(GLubyte *pixels, int w, int h,
                                      int wr, int hr, float scale)
{
  const float oneOver255 = 1.0/255.0;
  int i, j;
  Normal *nmap;
  float sqlen, reciplen, nx, ny, nz, c, cx, cy, dcx, dcy;

  nmap = malloc(sizeof(Normal)*w*h);
  if (nmap == NULL) {
    return NULL;  /* Memory allocation failure. */
  }

  for (i=0; i<h; i++) {
    for (j=0; j<w; j++) {
      /* Expand [0,255] texel values to the [0,1] range. */
      c = pixels[i*wr + j] * oneOver255;
      /* Expand the texel to its right. */
      cx = pixels[i*wr + (j+1)%wr] * oneOver255;
      /* Expand the texel one up. */
      cy = pixels[((i+1)%hr)*wr + j] * oneOver255;
      dcx = scale * (c - cx);
      dcy = scale * (c - cy);

      /* Normalize the vector. */
      sqlen = dcx*dcx + dcy*dcy + 1;
      reciplen = 1.0/sqrt(sqlen);
      nx = dcy*reciplen;
      ny = -dcx*reciplen;
      nz = reciplen;
      /* Repack the normalized vector into an RGB unsigned byte
         vector in the normal map image. */
      nmap[i*w+j].nx = 128 + 127*nx;
      nmap[i*w+j].ny = 128 + 127*ny;
      nmap[i*w+j].nz = 128 + 127*nz;
      /* The highest resolution mipmap level always has a
         unit length magnitude. */
      nmap[i*w+j].mag = 255;
    }
  }
  return nmap;
}
```

F LOADING AND MIPMAPPING A NORMAL MAP

This appendix shows how to load a normal map constructed by the `convertHeightFieldToNormalMap` shown in Appendix E. The `convertHeightFieldAndLoadNormalMapTexture` routine calls `convertHeightFieldToNormalMap` to generate a normal map image. This image is passed to OpenGL as the base level of a 2D texture. Then this image is recursively down-sampled and used to generate all of the normal map texture's mipmap levels. The down sampling process is careful to range-expand, average, renormalize, and range-compress the RGB vector on each down sampling and also keeps track of the magnitude of the successive denormalizations of the vector in the alpha component.

```
/* Given a normal map, create a down-sampled version of the normal map
   at half the width and height.  Use a 2x2 box filter to create each
   down-sample.  gluBuild2DMipmaps is not suitable because each down-sampled
   texel must also be renormalized. */
Normal *downSampleNormalMap(Normal *old, int w2, int h2, int w, int h)
{
  const float oneOver127 = 1.0/127.0;
  const float oneOver255 = 1.0/255.0;
```

```
Normal *nmap;
float x, y, z, l, invl;
float mag00, mag01, mag10, mag11;
int i, j, ii, jj;

/* Allocate space for the down-sampled normal map level. */
nmap = malloc(sizeof(Normal)*w*h);
if (NULL) {
  return NULL;  /* Memory allocation failure. */
}

for (i=0; i<h2; i+=2) {
  for (j=0; j<w2; j+=2) {

    /* The "%w2" and "%h2" modulo arithmetic makes sure that
       Nx1 and 1xN normal map levels are handled correctly. */

    /* Fetch the magnitude of the four vectors to be down-sampled. */
    mag00 = oneOver255 * old[ (i  )    *w2 +  (j  )    ].mag;
    mag01 = oneOver255 * old[ (i  )    *w2 + ((j+1)%h2)].mag;
    mag10 = oneOver255 * old[((i+1)%w2)*w2 +  (j  )    ].mag;
    mag11 = oneOver255 * old[((i+1)%w2)*w2 + ((j+1)%h2)].mag;

    /* Sum 2x2 footprint of red component scaled back to [-1,1] range. */
    x =  mag00 * (oneOver127 * old[ (i  )    *w2 +  (j  )    ].nx - 1.0);
    x += mag01 * (oneOver127 * old[ (i  )    *w2 + ((j+1)%h2)].nx - 1.0);
    x += mag10 * (oneOver127 * old[((i+1)%w2)*w2 +  (j  )    ].nx - 1.0);
    x += mag11 * (oneOver127 * old[((i+1)%w2)*w2 + ((j+1)%h2)].nx - 1.0);

    /* Sum 2x2 footprint of green component scaled back to [-1,1] range. */
    y =  mag00 * (oneOver127 * old[ (i  )    *w2 +  (j  )    ].ny - 1.0);
    y += mag01 * (oneOver127 * old[ (i  )    *w2 + ((j+1)%h2)].ny - 1.0);
    y += mag10 * (oneOver127 * old[((i+1)%w2)*w2 +  (j  )    ].ny - 1.0);
    y += mag11 * (oneOver127 * old[((i+1)%w2)*w2 + ((j+1)%h2)].ny - 1.0);

    /* Sum 2x2 footprint of blue component scaled back to [-1,1] range. */
    z =  mag00 * (oneOver127 * old[ (i  )    *w2 +  (j  )    ].nz - 1.0);
    z += mag01 * (oneOver127 * old[ (i  )    *w2 + ((j+1)%h2)].nz - 1.0);
    z += mag10 * (oneOver127 * old[((i+1)%w2)*w2 +  (j  )    ].nz - 1.0);
    z += mag11 * (oneOver127 * old[((i+1)%w2)*w2 + ((j+1)%h2)].nz - 1.0);

    /* Compute length of the (x,y,z) vector. */
    l = sqrt(x*x + y*y + z*z);
    if (l == 0.0) {
      /* Ugh, a zero length vector.  Avoid division by zero and just
         use the unperturbed normal (0,0,1). */
      x = 0.0;
      y = 0.0;
      z = 1.0;
    } else {
      /* Normalize the vector to unit length. */
      invl = 1.0/l;
      x = x*invl;
      y = y*invl;
      z = z*invl;
    }
    ii = i >> 1;
    jj = j >> 1;

    /* Pack the normalized vector into an RGB unsigned byte vector
       in the down-sampled image. */
    nmap[ii*w+jj].nx = 128 + 127*x;
    nmap[ii*w+jj].ny = 128 + 127*y;
    nmap[ii*w+jj].nz = 128 + 127*z;
```

```
      /* Store the magnitude of the average vector in the alpha
         component so we keep track of the magnitude. */
      l = l/4.0;
      if (l > 1.0) {
        nmap[ii*w+jj].mag = 255;
      } else {
        nmap[ii*w+jj].mag = 255*l;
      }
    }
  }
  free(old);
  return nmap;
}

/* Convert the supplied height field image into a normal map (a normalized
   vector range-compressed to the [0,1] range in RGB and A=1.0).  Load the
   base texture level, then recursively down-sample and load successive
   normal map levels (being careful to expand, average, renormalize,
   and unexpand each RGB value and also accumulate the average vector
   shortening in alpha). */
void convertHeightFieldAndLoadNormalMapTexture(GLubyte *pixels, int w, int h,
                                               int wr, int hr, float scale)
{
  Normal *nmap;
  int level, nw, nh;

  nmap = convertHeightFieldToNormalMap(pixels, w, h, wr, hr, scale);

  level = 0;
  /* Load original maximum resolution normal map. */
  /* The BGRA color component ordering is fastest for NVIDIA. */
  glTexImage2D(GL_TEXTURE_2D, level, GL_RGBA8, w, h, level,
    GL_BGRA_EXT, GL_UNSIGNED_BYTE, &nmap->nz);

  /* Down-sample the normal map for mipmap levels down to 1x1. */
  while (w > 1 || h > 1) {
    level++;

    /* Half width and height but not beyond one. */
    nw = w >> 1;
    nh = h >> 1;
    if (nw == 0) nw = 1;
    if (nh == 0) nh = 1;
    nmap = downSampleNormalMap(nmap, w, h, nw, nh);

    glTexImage2D(GL_TEXTURE_2D, level, GL_RGBA8, nw, nh, 0,
                 GL_BGRA_EXT, GL_UNSIGNED_BYTE, &nmap->nz);

    /* Make the new width and height the old width and height. */
    w = nw;
    h = nh;
  }
  free(nmap);
}
```

G CONFIGURE THE REGISTER COMBINERS FOR AMBIENT AND DIFFUSE ILLUMINATION

This appendix shows how to configure the register combiners to compute the ambient and diffuse illumination for the bump-mapping technique as described in Section 5.4.

```
void configCombinersForAmbientAndDiffusePass(void)
{
  GLfloat Iambient[4] = {0.1, 0.1, 0.1, 0.0};  /* Ambient illumination. */
  GLfloat Kdiffuse[4] = {0.9, 1.0, 1.0, 0.0};  /* Diffuse material characteristic. */

  glCombinerParameteriNV(GL_NUM_GENERAL_COMBINERS_NV, 2);
```

```
glCombinerParameterfvNV(GL_CONSTANT_COLOR0_NV, Iambient);
glCombinerParameterfvNV(GL_CONSTANT_COLOR1_NV, Kdiffuse);

/*** GENERAL Combiner ZERO, RGB portion. ***/
/* Argb = 3x3 matrix column1 = expand(texture0rgb) = N' */
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
  GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV, GL_RGB);
/* Brgb = expand(texture1rgb) = L */
glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
  GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_RGB);

/* spare0rgb = Argb dot Brgb
             = expand(texture0rgb) dot expand(texture1rgb) = L dot N' */
glCombinerOutputNV(GL_COMBINER0_NV, GL_RGB,
  GL_SPARE0_NV, GL_DISCARD_NV, GL_DISCARD_NV,
  GL_NONE, GL_NONE, GL_TRUE, GL_FALSE, GL_FALSE);

/*** GENERAL Combiner ZERO, Alpha portion. ***/
/* Aa = 1 */
glCombinerInputNV(GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_A_NV,
  GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA);
/* Ba = expand(texture1b) = Lz */
glCombinerInputNV(GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_B_NV,
  GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_BLUE);
/* Ca = 1 */
glCombinerInputNV(GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_C_NV,
  GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA);
/* Da = expand(texture1b) = Lz */
glCombinerInputNV(GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_D_NV,
  GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_BLUE);

/* spare0a = 4*(1*Lz + 1*Lz) = 8*expand(texture1b) */
glCombinerOutputNV(GL_COMBINER0_NV, GL_ALPHA,
  GL_DISCARD_NV, GL_DISCARD_NV, GL_SPARE0_NV,
  GL_SCALE_BY_FOUR_NV, GL_NONE, GL_FALSE, GL_FALSE, GL_FALSE);

/*** GENERAL Combiner ONE, RGB portion. ***/
/* Argb = spare0rgb = L dot N' */
glCombinerInputNV(GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
  GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
/* Brgb = expand(texture0a) = normal map denormalization factor */
glCombinerInputNV(GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
  GL_TEXTURE0_ARB, GL_UNSIGNED_IDENTITY_NV, GL_ALPHA);

/* spare0rgb = Argb Brgb = L dot N' scaled by the normal map denormalization factor */
glCombinerOutputNV(GL_COMBINER1_NV, GL_RGB,
  GL_SPARE0_NV, GL_DISCARD_NV, GL_DISCARD_NV,
  GL_NONE, GL_NONE, GL_FALSE, GL_FALSE, GL_FALSE);

/*** GENERAL Combiner ONE, Alpha portion. ***/
/* Discard all outputs. */
glCombinerOutputNV(GL_COMBINER1_NV, GL_ALPHA,
  GL_DISCARD_NV, GL_DISCARD_NV, GL_DISCARD_NV,
  GL_NONE, GL_NONE, GL_FALSE, GL_FALSE, GL_FALSE);

/*** FINAL Combiner. ***/
/* A = spare0a = per-pixel self-shadowing term */
glFinalCombinerInputNV(GL_VARIABLE_A_NV,
  GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_ALPHA);
/* B = EF */
glFinalCombinerInputNV(GL_VARIABLE_B_NV,
  GL_E_TIMES_F_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
/* C = zero */
glFinalCombinerInputNV(GL_VARIABLE_C_NV,
  GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
```

```
    /* D = C0 = ambient illumination contribution */
    glFinalCombinerInputNV(GL_VARIABLE_D_NV,
      GL_CONSTANT_COLOR0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* E = C1 = diffuse material characteristic */
    glFinalCombinerInputNV(GL_VARIABLE_E_NV,
      GL_CONSTANT_COLOR1_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* F = spare0rgb = diffuse illumination contribution = L dot N' */
    glFinalCombinerInputNV(GL_VARIABLE_F_NV,
      GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* diffuse RGB color  = A*E*F + D = diffuse modulated by self-shadowing term and the
       diffuse material characteristic + ambient */

    /* G = spare0a = self-shadowing term = 8*expand(texture1b)  */
    glFinalCombinerInputNV(GL_VARIABLE_G_NV,
      GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_ALPHA);

    glEnable(GL_REGISTER_COMBINERS_NV);
  }
```

H CONFIGURE THE REGISTER COMBINERS FOR SPECULAR ILLUMINATION

This appendix shows how to configure the register combiners to compute the specular illumination for the bump-mapping technique as described in Section 5.5.

```
  void configCombinersForSpecular(void)
  {
    glCombinerParameteriNV(GL_NUM_GENERAL_COMBINERS_NV, 2);

    /*** GENERAL Combiner ZERO, RGB portion. ***/
    /* Argb = 3x3 matrix column1 = expand(texture0rgb) = N' */
    glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_A_NV,
      GL_TEXTURE0_ARB, GL_EXPAND_NORMAL_NV, GL_RGB);
    /* Brgb = expand(texture1rgb) = H */
    glCombinerInputNV(GL_COMBINER0_NV, GL_RGB, GL_VARIABLE_B_NV,
      GL_TEXTURE1_ARB, GL_EXPAND_NORMAL_NV, GL_RGB);

    /* spare0rgb = Argb dot Brgb = expand(texture0rgb) dot expand(texture1rgb)
                 = N' dot H */
    glCombinerOutputNV(GL_COMBINER0_NV, GL_RGB,
      GL_SPARE0_NV, GL_DISCARD_NV, GL_DISCARD_NV,
      GL_NONE, GL_NONE, GL_TRUE, GL_FALSE, GL_FALSE);

    /*** GENERAL Combiner ZERO, Alpha portion. ***/
    /* Aa = texture1b = unexpanded Hz */
    glCombinerInputNV(GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_A_NV,
      GL_TEXTURE1_ARB, GL_UNSIGNED_IDENTITY_NV, GL_BLUE);
    /* Ba = 1 */
    glCombinerInputNV(GL_COMBINER0_NV, GL_ALPHA, GL_VARIABLE_B_NV,
      GL_ZERO, GL_UNSIGNED_INVERT_NV, GL_ALPHA);

    /* spare0a = 1 * texture1b = unexpanded Hz */
    glCombinerOutputNV(GL_COMBINER0_NV, GL_ALPHA,
      GL_SPARE0_NV, GL_DISCARD_NV, GL_DISCARD_NV,
      GL_NONE, GL_NONE, GL_FALSE, GL_FALSE, GL_FALSE);

    /*** GENERAL Combiner ONE, RGB portion. ***/
    /* Argb = 0 */
    glCombinerInputNV(GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_A_NV,
      GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* Brgb = 0 */
    glCombinerInputNV(GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_B_NV,
      GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* Crgb = spare0rgb = H dot N' */
    glCombinerInputNV(GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_C_NV,
      GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* Drgb = spare0rgb = H dot N' */
    glCombinerInputNV(GL_COMBINER1_NV, GL_RGB, GL_VARIABLE_D_NV,
```

```
      GL_SPARE0_NV, GL_SIGNED_IDENTITY_NV, GL_RGB);

    /* spare0rgb = ((spare0a >= 0.5) ? spare0rgb^2 : 0)
                 = ((H dot N > 0) ? (H dot N')^2 : 0) */
    glCombinerOutputNV(GL_COMBINER1_NV, GL_RGB,
      GL_DISCARD_NV, GL_DISCARD_NV, GL_SPARE0_NV,
      GL_NONE, GL_NONE, GL_FALSE, GL_FALSE, GL_TRUE);

    /*** GENERAL Combiner ONE, Alpha portion. ***/
    /* Discard all outputs. */
    glCombinerOutputNV(GL_COMBINER1_NV, GL_ALPHA,
      GL_DISCARD_NV, GL_DISCARD_NV, GL_DISCARD_NV,
      GL_NONE, GL_NONE, GL_FALSE, GL_FALSE, GL_FALSE);

    /*** FINAL Combiner. ***/
    /* A = EF */
    glFinalCombinerInputNV(GL_VARIABLE_A_NV,
      GL_E_TIMES_F_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* B = EF */
    glFinalCombinerInputNV(GL_VARIABLE_B_NV,
      GL_E_TIMES_F_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* C = zero */
    glFinalCombinerInputNV(GL_VARIABLE_C_NV,
      GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* D = zero = no extra specular illumination contribution */
    glFinalCombinerInputNV(GL_VARIABLE_D_NV,
      GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* F = spare0rgb = (H dot N')^2 */
    glFinalCombinerInputNV(GL_VARIABLE_E_NV,
      GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* F = spare0rgb = (H dot N')^2 */
    glFinalCombinerInputNV(GL_VARIABLE_F_NV,
      GL_SPARE0_NV, GL_UNSIGNED_IDENTITY_NV, GL_RGB);
    /* specular RGB color = A*B = (E*F)*(E*F) = (H dot N')^8 */

    /* G = 0 */
    glFinalCombinerInputNV(GL_VARIABLE_G_NV,
      GL_ZERO, GL_UNSIGNED_IDENTITY_NV, GL_ALPHA);

    glEnable(GL_REGISTER_COMBINERS_NV);
  }
```