Hardware Accelerated Per-Pixel Shading

Gerald Schröcker
gerald@schroecker.info

Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

Today graphics accelerators are rapidly becoming programmable. This allows to write custom shading algorithms which evaluate a shading equation per-pixel in real-time. In this paper we outline the main features of the used graphics hardware, then we describe the techniques to achieve high-quality local illumination using Phong shading. Additionaly we integrate bump mapping as a technique for simulating the effect of light reflecting from small surface perturbations to enhance the realism without increasing the geometric complexity.

KEYWORDS: Per-Pixel Lighting, Phong Shading, Bump Mapping, GeForce3

1 Introduction

Until recently, the major concern in the development of new graphics hardware has been to increase the performance of the traditional rendering pipeline. Today, graphics accelerators with a performance of several million textured, lit triangles per second are within reach even for the low end. As a consequence, the focus is beginning to shift away from higher performance towards higher quality renderings and an increased feature set [8]. Traditionally hardware renderers only support the Phong [16] lighting model in combination with Gouraud [7] shading, which means that the actual Phong lighting equation is evaluated per vertex, and simple linear interpolations is used for shading pixels. This method is relatively easy to implement in hardware, but for a moderately tesselated surface the drawbacks of Gouraud shading as diffused, crawling highlights and Mach banding become evident [15]. However, consumer-level graphics hardware is rapidly becoming programmable at both the vertex processing and the fragment processing stage. This allows to write custom shading algorithms – so-called *shaders* – which evaluate a shading program per-pixel in real-time. These shading algorithms can more realistically model the enormous variety of materials and lighting effects that exist in the real world. In this paper we will develop shading techniques which run in one rendering pass on advanced graphics accelerators.

2 Survey

2.1 Lighting Models

The first lighting model which accounts for diffuse surfaces was developed by Gouraud [7]. In 1975 Phong [16] proposed the first model in computer graphics able to deal with non-diffuse surfaces. In this model the color of a pixel is expressed as a linear combination of a diffuse part and a specular part. Blinn improved the physical correctness of this model and made it visually more satisfying. This so-called Blinn-Phong model [2] is commonly used for hardware accelerated lighting. These early models were ad hoc empirical models without any exact value of energy or intensity. The most important of physically more correct models is the one by Cook and Torrance [4], it is based on a Gaussian micro facet distribution. In addition to isotropic models, anisotropic models like the model from Banks [1] have also been proposed. As mentioned above, hardware based rendering methods usually use the Blinn-Phong model [2], because of its mathematical simplicity. However, Heidrich et al. [8] recently showed how to handle more complex BRDF models by factoring the Banks model and the Cook-Torrance model analytically, and storing the factors in texture maps. The original model can then be reconstructed using texture mapping. Kautz et al. [9] and McCool et al. [13] reparameterized BRDFs and then decomposed them numerically. Again the original model is reconstructed using texture mapping.

2.2 Bump Mapping

Bump mapping was originally introduced by Blinn [3]. It has recently found its way into hardware-accelerated rendering. The first technique that worked on consumer graphics hardware is called *texture embossing*, it is a multipass method that is quite limited. *Dot product bump mapping* [10] is a better method, it directly stores the normals of the surface in texture maps, but needs advanced hardware features, namely the ability to compute per-pixel dot-products. Graphics hardware researchers have also proposed a variety of approaches for implementing bump mapping through dedicated hardware schemes. Evans & Sutherland [5] and SGI [15] have both proposed bump mapping approaches for high-end graphics hardware. The SGI technique introduced lighting in *tangent space*, which is also used in this paper.

3 Hardware Architecture

For real-time rendering a completely programmable graphics solution is impractical – at least for now. Therefore programmability is enabled in three interesting areas, while leaving the rest of the graphics pipeline running as it always has. Until now these features are optional extensions to the OpenGL standard [18] and thus hardware dependent. Since every manufacturer of graphics hardware defines its

own extensions[1], we will restrict our description to graphics boards with Nvidias GeForce3 processor.

3.1 Vertex Shader

A vertex shader [12] is a small assembly-language program. When enabled, it replaces the transform- and lighting-computations of the fixed-function pipeline. A vertex shader operates on a single vertex at a time. It does not operate on primitives and is unable to generate new vertices. Frustum clipping, perspective divide and viewport transformation are left to fixed processing stages. The computation-model of vertex shaders is straightforward. For every vertex to be processed, the vertex shader executes its program. It has access to four different types of memory locations: the per-vertex data of an incoming vertex, constant memory, temporary registers, and per-vertex output-registers (Figure 1). The basic data type is the

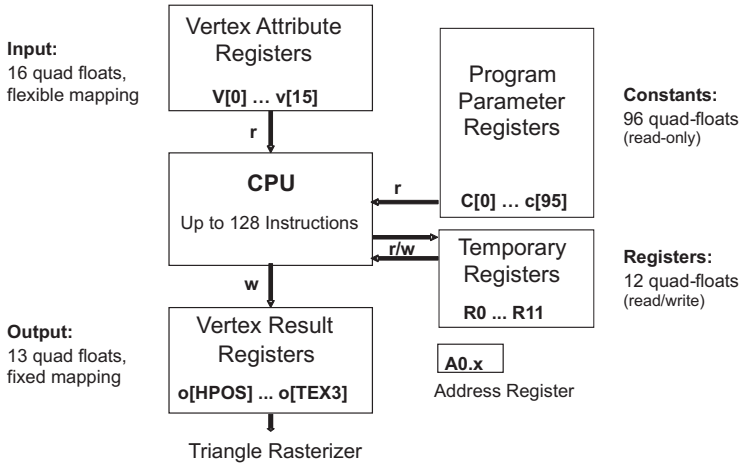


Figure 1: Vertex Program

quad-float vector. In order to deal with efficient scalar packing and extraction, the input vectors can have their components arbitrarily rearranged or replicated (swizzled), output writes have a component write mask. The instruction set can be divided into vector, scalar and miscellaneous operations. No branching, jumping or looping is supported to maintain pipeline efficiency. All instructions have the same latency, this limits the complexity of any instruction but improves programmability and simplifies the hardware.

3.2 Texture Shader

Texture shaders [11] provide a superset of conventional OpenGL texture addressing [18]. They expose a number of operations that can be used to compute texture coordinates per-fragment rather than using simple interpolated per-vertex coordinates. The shader operations belong to four main categories: 1. *Conventional*

[1]the upcoming OpenGL 2.0 standard [17] will hopyfully remedy this situation

Texture Access, these are the standard 1D, 2D, 3D texture access modes, furthermore cube map textures and rectangular textures are supported. 2. *Dependent Texture Access*, these modes use the result from a previous texture stage to affect the lookup of the current stages. Dependent 2D scaling and biasing is possible. 3. *Dot Product Texture Access*, these operations calculate a high precision (float) dot product from texture coordinates and a vector derived from the results of a previous shader stage. The resulting scalar value is used for accessing a texture. 4. *Special Modes*, these operations cull the current fragment or convert the texture coordinates directly to colors without accessing a texture. In order to support the various per-pixel math that must be done in the texture shaders, a number of new texture formats for encoding vectors are introduced. Most notably signed texture formats and 16 bit high precision formats.

3.3 Register Combiners

In order to gain explicit control over per-fragment computation, Nvidia provides the register combiners extension [11]. With this extension enabled, the standard OpenGL texture environment is completely bypassed and substituted by a register-based unit. This unit consist of eight extremely flexible general combiner stages (Figure 2) and one final combiner stage. In a register combiner per-fragment in-

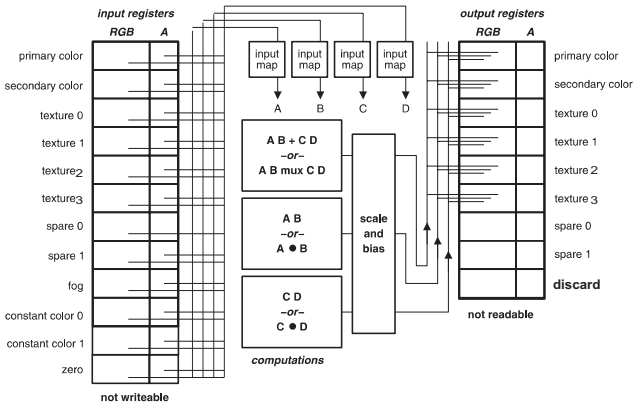


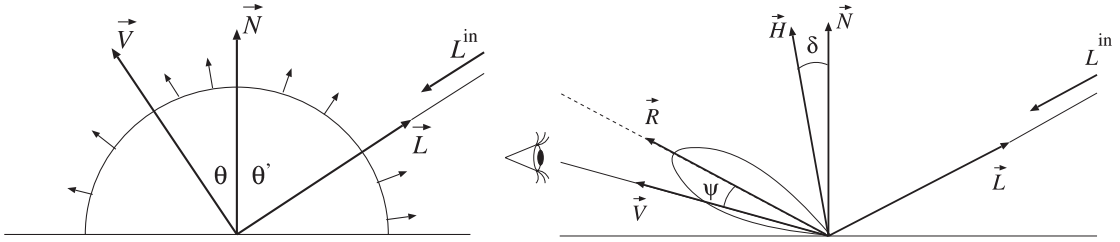
Figure 2: A general combiner stage supports arbitrary register mappings and complex arithmetic computations.

formation is stored in a set of input registers. The contents of these registers can be arbitrarily mapped to the four variables A, B, C and D. After combining these variables, e.g. by dot product ($A \cdot B$), the results are scaled and biased and finally written to arbitrary output registers. The output registers of the first combiner stage are then the input registers for the next stage. An additional feature is, that fixed point color components, which are usually clamped to a range of [0, 1] can internally be expanded to a signed range [-1,1]. This allows vector components to be stored in the color registers without the need to scale and bias them. The resulting fragment output from the final combiner stage is processed with the standard OpenGL per-fragment operations, like depth test or alpha-blending.

4 Implementation

4.1 Lighting Model

Two kinds of surfaces can be distinguished according to the way they reflect light. On one hand, there are *diffuse surfaces* for which light is reflected in every direction (Figure 3a). On the other hand, there are *specular surfaces* for which light is reflected only in a small area around the mirror direction (Figure 3b). There are



(a) Diffuse Reflection (b) Specular Reflection

Figure 3: Blinn-Phong Lighting Model

many lighting models to describe this behavior. For simplicity we will use the Blinn-Phong [2] lighting model (Formula 1).

$$I_{out} = I_{Light} k_d \max(0, \vec{N} \cdot \vec{L}) + I_{Light} k_s \max(0, \vec{N} \cdot \vec{H})^n \tag{1}$$

I_{Light} is the color of the light, k_d is the diffuse color, k_s is the specular color; n is the specular exponent which defines the shininess of the surface. $\vec{N}$ is the normalized surface normal, $\vec{L}$ is the normalized direction vector pointing to the light source. The half-angle vector $\vec{H}$ is the half-way unit vector between $\vec{L}$ and $\vec{V}$ defined in Formula 2.

$$\vec{H} = \frac{\vec{L} + \vec{V}}{\left|\vec{L} + \vec{V}\right|} \tag{2}$$

Where $\vec{V}$ is the normalized vector to the viewer. It is important to max out negative dot product terms, as a negative dot product indicates that the point is in shadow and receives no light. The key aspects for successful per-pixel lighting are to provide the needed vector parameters ($\vec{N}$, $\vec{L}$ and $\vec{H}$) for evaluating the lighting equation and to compute the per-pixel dot products. In the next Section we will see how to interpolate and normalize this vectors.

4.2 Parameter Interpolation and Normalization

Phong shading implies that for every pixel, the vectors being involved in the shading equation are interpolated, normalized and their dot product computed. Without normalization the highlights get lost across a polygon. To be general, the interpolation of two vectors $\vec{v}_1$ and $\vec{v}_2$ is considered. This can be any vectors: the light vector $\vec{L}$, viewing vector $\vec{V}$ or normal vector $\vec{N}$.

4.2.1 Spherical Interpolation

The dot product between two vectors is only equivalent to the cosine of the angle if the two vectors are unit vectors. So the two vectors $\vec{v}_1$ and $\vec{v}_2$ should be interpolated in a way that the interpolant $\vec{v}(t)$ is moving uniformly between the two vectors and its length remains one. As [14] point out this interpolations works on the surface of the unit sphere and therefore it is called *spherical interpolation* (Formula 3).

$$\vec{v}(t) = \frac{\sin(1-t)\alpha}{\sin\alpha}\vec{v}_1 + \frac{\sin(t)\alpha}{\sin\alpha}\vec{v}_2 \tag{3}$$

Where $\cos\alpha = \vec{v}_1 \cdot \vec{v}_2$. Unfortunately *spherical interpolation* is not available on current graphics accelerators, only *linear interpolation* is supported. When using linear interpolation this results in denormalized vectors (Figure 4) and wrong shading intensities.

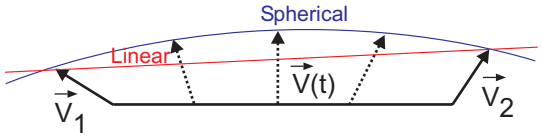
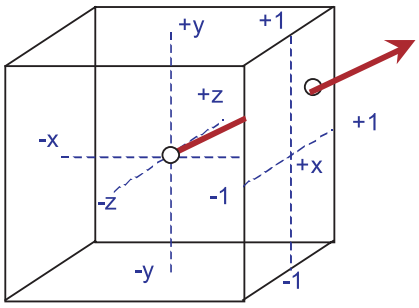


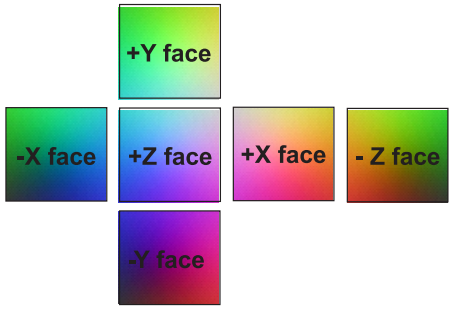
Figure 4: Linear Interpolation versus Spherical Interpolation

4.2.2 Cube-Map

One method to renormalize the vectors is to use a *cube map texture*. Cube map texturing is a form of texture mapping that uses a 3D direction vector built from the texture coordinates (s, t, r) to access a texture that consists of six square 2D images arranged like the faces of a cube [10]. In order to normalize a vector,



(a) Directional Access



(b) RGB Coded Normalization Faces

Figure 5: Cube Map

the cube map can be thought as a way to store a look-up table indexed by a direction vector (Figure 5a). This means that vectors of varying length which point in the same direction do not change the lookup result. The normalization

given in Formula 4 is precomputed for discrete values of $\vec{v}$ and stored in the cube map.

$$\vec{v}' = \frac{\vec{v}}{\|\vec{v}\|} = \frac{\vec{v}}{\sqrt{\vec{v} \cdot \vec{v}}} = \frac{\vec{v}}{\sqrt{\vec{v}_x^2 + \vec{v}_y^2 + \vec{v}_z^2}} \tag{4}$$

Since parts of the render pipeline work only with positive values, the signed vector components must be range compressed from $[-1, 1]$ to $[0, 1]$ and stored as colors (Figure 5b).

4.2.3 Register Combiners

A faster technique for normalization without using texture maps is to use the register combiners extension [11]. With register combiners only addition, subtraction, multiplication and dot products are possible (Section 3.3). Direct computation of the normalization equation shown in Formula 4 is not possible. Given that the vector $\vec{v}$ is derived from the interpolation of a unit-length vector across the polygon and the angle between the per-vertex vectors is not too big, this expression can be approximated by a Taylor series (Formula 5).

$$\vec{v}' = \vec{v}\left(1 - \frac{1}{2}(\vec{v} \cdot \vec{v} - 1) + \frac{3}{8}(\vec{v} \cdot \vec{v} - 1)^2 - \frac{5}{16}(\vec{v} \cdot \vec{v} - 1)^3 + \ldots\right) \tag{5}$$

Given the assumptions mentioned above and the limited 8 bit precision in the register combiners this expressions is truncated after the linear term (Formula 6).

$$\vec{v}' \cong \frac{\vec{v}}{2}(3 - \vec{v} \cdot \vec{v}) = \vec{v} + 0.5\vec{v}(1 - \vec{v} \cdot \vec{v}) \tag{6}$$

The last expression can be implemented directly with register combiner arithmetic in two general combiner stages.

4.3 Tangent Space

Lighting can be computed in an arbitrary 3D coordinate system as long as all vector parameters involved are oriented with respect to the same coordinate system [10]. This allows one to select the most convenient coordinate system for lighting. *Tangent space* is just such a local coordinate system. There are two reasons why lighting in tangent space is very efficient. The first one is that the normal vector $\vec{N}$ equals always $(0, 0, 1)$ in tangent space, therefore $\vec{N}$ needs no longer interpolated and normalized. The second reason is that in tangent space the *perturbed normal* for bump mapping can be read directly from a texture[2] as Kilgard [10] develop. The orthonormal basis for tangent space is formed by the surface normal $\vec{N}$, the surface tangent vector $\vec{T}$, and the binormal $\vec{B}$ defined as $\vec{N} \times \vec{T}$ (Figure 6) For the illumination calculation to proceed properly, the light and half-angle vectors are transformed into tangent space via a 3×3 matrix whose columns are $\vec{T}$, $\vec{B}$, and $\vec{N}$. The transformations of the light and half-angle vectors should be performed at

[2] as long as some conditions, mainly the so-called *square patch assumption* are observed [15, 10]

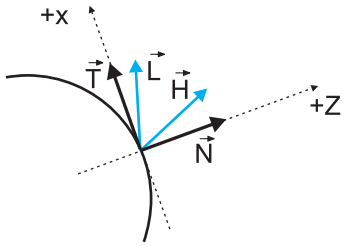
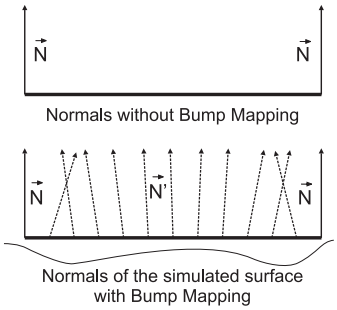


Figure 6: Definition of Tangent Space

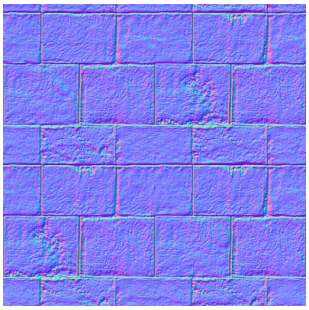
every pixel; however, if the change of the local tangent space across a polygon is small, a good approximation can be obtained by transforming the vectors only at the polygon vertices. They are then interpolated and normalized in the polygon interior. Therefore tangent space is constructed efficiently on a per-vertex basis with the help of a vertex shader (Section 3.1).

4.4 Bump Mapping

Until now we have almost exclusively treated Phong shading. An additional way to add more realism to a rendered scene is to use *bump mapping*. It allows to increase the visual detail of a scene without requiring excessive amounts of geometric detail. It is a technique that was invented by Blinn [3] to add roughness or wrinkles to a smooth surface. It does not change the underlying geometry of the model, but fools the shading to produce an interesting surface by using a perturbed surface normal $\vec{N}'$ read from a normal map (Figure 7a). Bump mapping is a very 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 large-scale object shape required for efficient transformation, rasterization and hidden surface removal [10]. Since lighting is already



(a) Perturbed Normals from the Normal Map



(b) Color Coded Normal Map

Figure 7: Bump Mapping

done in tangent space – where the large-scale normal $\vec{N}$ is always $(0, 0, 1)$ – the perturbed small-scale surface normal $\vec{N}'$ can be read directly from a texture map. This so-called normal map can be constructed from a height map by using finite

differences to get the local tangent plane and the corresponding surface normal (Figure 7b).

4.5 Self Shadowing

With bump mapping, there are actually *two* surface normals that should be considered for lighting. The unperturbed normal $\vec{N}$ is based on the surface's large-scale geometry, while the perturbed normal $\vec{N}'$ is based on the small-scale structure. Either normal can create self-shadowing situations. Figure 8 shows a situation where the perturbed normal $\vec{N}'$ is subject to illumination. However, the point on the surface should not receive illumination from the light because the unperturbed normal $\vec{N}$ indicates that the point is in shadow due to the large-scale geometry [10]. In

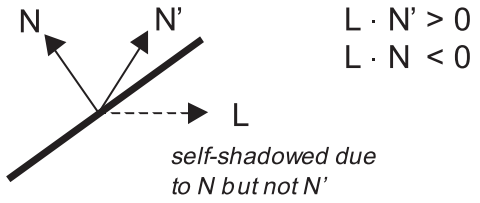


Figure 8: Self Shadowing

order to account for self-shadowing due to the perturbed surface normal and the unperturbed normal the lighting equation 1 should be rewritten as in Equation 7.

$$I_{out} = I_{Light} s_{self} k_d \max(0, \vec{N}' \cdot \vec{L}) + I_{Light} s_{self} k_s \max(0, \vec{N}' \cdot \vec{H})^n \qquad (7)$$

where

$$s_{self} = \begin{cases} 1 & \vec{L} \cdot \vec{N} > 0 \\ 0 & \vec{L} \cdot \vec{N} \leq 0 \end{cases}$$

Without this extra level of clamping, bump-mapped surfaces can show false illumination artifacts in otherwise dark regions. In practice, the step function of s_{self} shown above can lead to temporal aliasing artifacts, as pixel along the self-shadowing boundary may pop on and off abruptly. Therefore it is better to replace the step function by a steep ramp [10].

5 Results

5.1 Normalization

Figure 9 shows the effect of different vector normalization methods. So as to avoid precision problems the specular exponent is relatively small ($n = 8$). Gouraud shading to the left is only shown for reference purposes. Without normalization it is clearly visible, that the shape of the highlight and intensity are not correct. With register combiner normalization the results are astonishingly good, although that only a linear square root approximation is used for normalization. The visible banding effects are primary caused by the limited precision for the exponentiation of the

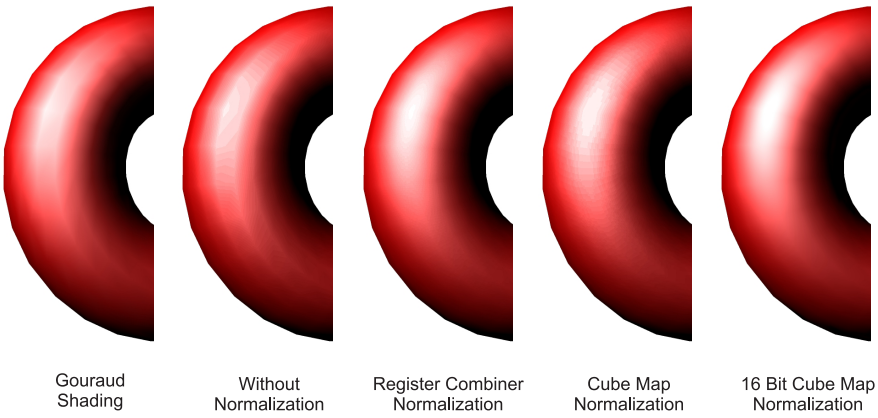


Figure 9: Quality of Different Normalization Methods

specular intensity. 8 bit cube map normalization works not as good as expected. One cause is that one more bit in comparison to register combiner normalization gets lost as the whole range $[-1, 1]$ must be range compressed to $[0, 255]$ (register combiners work internally with 8+1 bits so the range is $[-255, 255]$). The best render quality is reached by using a 16 bit normalization cube map in combination with a high precision dot product in the texture shaders.

5.2 Precision

Computations in the register combiners are performed in 8 bit (+ 1 sign bit) fixed point, this results in precision and dynamic range problems as shown in Figure 10. In order to isolate precision problems from interpolation and normalization issues the used sphere model is highly tesselated. In the shader with register combiner

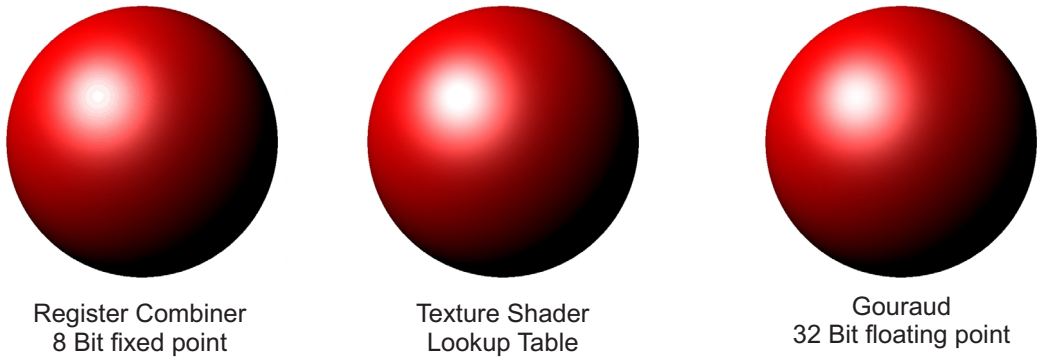


Figure 10: Comparison of Precision Problems

normalization, the exponentiation is computed by repeated self multiplication, for higher specular exponents this results in visible banding. The shader with table lookup performs much better. In this shader the result of a high precision dot product between $\vec{N}$ and $\vec{H}$ is used to access a texture where the specular intensity is stored. These values are stored as 8 bit values, but through the use of linear interpolation between the table values the quality is very good. The standard

Gouraud shader is used only for reference purpose as no per-pixel shading takes place. It should be noted, that in the inner zone of the highlight the Gouraud shader performs still better than the texture shader, because the exponentiation function gets very steep near one and there are too few values in the table to interpolate perfectly [6].

5.3 Bump Mapping

After the qualitative comparison of different implementation variants, we will now examine how these quality differences show up with activated decal and bump map textures. Unfortunately bump mapping or even texture mapping is not possible for shaders with 16 bit cube map normalization as all four texture units are occupied by this task. Surprisingly the different normalization techniques and precision issues shown in the preceding two Sections have very little impact on the obtained image quality when bump mapping is used (Figure 11). We assume that the dif-

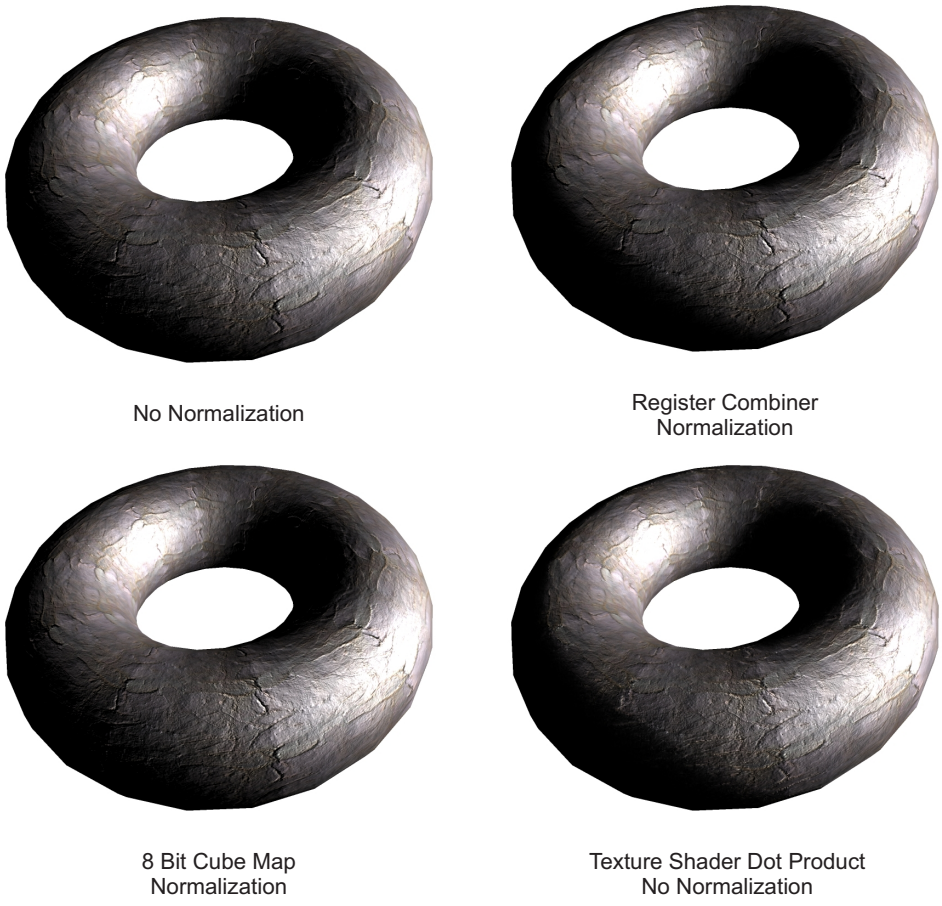


Figure 11: Comparison of Different Bump Map Methods

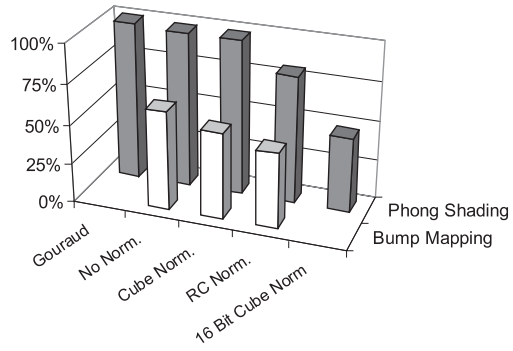
ferent surface normals read from the normal map result in rapid changing lighting intensities and therefore the less perfect shading qualities get lost in noise.

5.4 Timing

Finally we compare the render speed of different shaders. All timings have been done for a screen-filling mesh with 20k triangles (results for other triangle counts are similar). We would like to point out some particular results from Table 12a.

Shader	FPS		FPS	
	abs.	rel.	abs.	rel.
Gouraud	175	100%	-	-
No Norm.	172	98%	110	63%
Cube Norm.	172	98%	96	55%
RC Norm.	141	81%	84	48%
16 Bit Cube Norm	83	47%	-	-
Used Textures	Phong Shading		Bump Mapping	

(a) Render Time Table



(b) Render Time Chart

Figure 12: Comparison of Shader Render Times

For simple Phong shading without bump mapping or texturing, the shader with register combiner normalization runs at 80% of the speed of standard Gouraud shading, although much better visual quality is obtained (Figure 9). The shader with 16 bit cube map normalization and texture shader dot product which delivers the best image quality (Figure 10) runs with about half the speed of Gouraud shading. If shading with a decal texture and bump mapping is enabled, the shader with register combiner normalization runs also with slightly less than half the speed of standard Gouraud shading although the visual detail and shading quality are much better (Figure 11).

6 Conclusion and Future Work

Vertex programs, texture shaders and register combiners provide a way to take control of the graphics pipeline at various stages. All these pieces together enable high quality real-time per-pixel shading. In the future one can expect to get even closer to a fully hardware-accelerated RenderMan [19] like programmable shading. Despite the great technological progress, a few drawbacks make it difficult to develop high-quality shaders: it would be very helpful if length preserving specular interpolation would be available, since the currently supported linear interpolation leads to a number of problems and makes expensive per-pixel normalization necessary. The register combiners are a possible way to implement programmable per-pixel operations, but a more flexible shader definition language as proposed in the upcoming OpenGL 2.0 standard [17] would be helpful.

In our current work, the vectors used to construct the tangent space are derived from the analytical definition of the used surface. In order to make the promising results for high-quality per-pixel shading applicable to a wide range of applications these vectors should be computed from an ordinary triangle mesh. Another

reasonable extension would be to apply a more realistic shading model than the currently used Blinn-Phong [2] shading model. For example the Cook-Torrance [4] model or even true BRDF distributions [13] could be used. Kautz [9] describe a very promising approach which unfortunately needs too many render passes on current hardware.

References

[1] David C. Banks. Illumination in diverse codimensions. In *Proceedings of SIGGRAPH '94*, pages 327–334, 1994.

[2] James F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics*, 11(2):192–198, July 1977.

[3] James F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics*, 12(3):286–292, August 1978.

[4] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. volume 15, pages 307–316, August 1981.

[5] Muchael Cosman and Robert Grange. Cig scene realism: The world tomorrow. In *Proceedings of I/ITSEC*, page pp. 628, 1996.

[6] Andrej Ferko, Markus Grabner, Anton Mateášik, Gerald Schröcker, and Marek Zimányi. On phong's model alternatives.

[7] Henri Gouraud. Computer display of curved surfaces. *IEEE Trans. Computers*, C-20(6):623–629, 1971.

[8] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Siggraph 1999, Annual Conference Proceedings*, 1999.

[9] Jan Kautz and Hans-Peter Seidel. Towards interactive bump mapping with anisotropic shift-variant BRDFs. In *Proceedings of the 2000 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 2000.

[10] Mark J. Kilgard. A practical and robust bump-mapping technique for today's gpus. In *GDC 2000: Advanced OpenGL Game Development*, July 2000.

[11] Mark J. Kilgard. *NVIDIA OpenGL Extension Specifications*, 11 2001.

[12] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *SIGGRAPH 2001, Computer Graphics Proceedings*, 2001.

[13] Michael D. McCool, Jason Ang, and Anis Ahmad. Homomorphic factorization of BRDFs for high-performance rendering. In *SIGGRAPH 2001, Computer Graphics Proceedings*, 2001.

[14] Abbas Ali Mohamed. Hardware implementation of phong shading using spherical interpolation. Technical report, Budapest University of Technology and Economics, 2001.

[15] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *SIGGRAPH 97 Conference Proceedings*, pages 303–306, 1997.

[16] Bui-Tuong Phong. Illumination for computer generated pictures. *CACM June 1975*, 18(6):311–317, 1975.

[17] John Schimpf. Opengl 2.0 whitepaper . `http://www.3dlabs.com/support/developer/ogl2/`.

[18] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification (version 1.3). Technical report, Silicon Graphics, Inc., 2001.

[19] Steve Upstill. *The RenderMan Companion.* Addison-Wesley, Reading, MA, USA, 1990.