A Hardware Accelerator for the Specular Intensity of Phong Illumination Model in 3-Dimensional Graphics

Young-Su Kwon, In-Cheol Park and Chong-Min Kyung

Department of Electrical Engineering, KAIST,
Kusong-dong, Yousong-gu, Taejon 305-701, Korea
Tel : +82-42-869-3461
e-mail : {yskwon,icpark,kyung}@duo.kaist.ac.kr

Abstract— **This paper presents a special hardware implementation developed for the computation of the specular term which is the most time consuming part in the Phong's illumination. In the Phong shading, the exponentiation operation of two floating-point numbers is necessary for each point inside a polygon. An approximation algorithm is developed to speed up the exponentiation operation, and it is supported by simple hardware that can be easily merged into a floating-point multiplier. The exponentiation operation takes just 4 cycles in the proposed hardware while it takes about 100-200 cycles in conventional floating-point units. Although an approximation algorithm is employed for the exponentiation operation, the amount of error is so minimal that the difference is virtually indistinguishable.**

I. INTRODUCTION

In computer graphics, polygon meshes are widely used to approximate curved surfaces. These polygon meshes must be adequately shaded to make the underlying geometry smooth. Two commonly used shading methods are Gouraud shading[2] and Phong shading[1].

The specular highlights are significant to give an object the visual cues about surface geometry and properties. It is well known that the specular highlights are completely missed or distorted in the Gouraud shading. However, the Phong shading represents specular highlights very well because the Phong illumination equation is calculated at every pixel to include the specular reflection term.

In spite of this, most 3D graphics accelerators are based on the Gouraud shading due to the computational cost of the Phong shading. Even the fastest high quality graphics workstations like the recently announced InfiniteReality[3] are still based on the Gouraud shading. According to rapid improvements in VLSI technologies and CAD tools supporting chip designs, real-time Phong shading will be the next technology-push in computer graphics.

The most time consuming part in the Phong illumination is to compute the specular term which requires the exponentiation of two floating-point numbers. It takes a very long CPU time to get the accurate exponentiation result. To do real-time Phong shading and to implement practical Phong shading hardware, it is essential to develop a fast algorithm that computes the specular term effectively.

This paper describes an approximation algorithm for computing the exponentiation of two floating-point numbers. The proposed algorithm can be implemented easily by doing a simple extension of floating-point multipliers.

This paper is organized as follows. In section II, the Phong illumination model and the important operations in the specular term computation are explained. In section III, the previous ways to compute the specular term are explained. In section IV, an efficient method is presented, and the experimental results are shown in section V.

II. THE PHONG ILLUMINATION MODEL

The Phong shading is often used to shade planar polygonal approximated surfaces smoothly[1]. In this shading model, the intensity I at an internal point of a polygon depends on the reflectance of the objects under consideration. For each pixel, the normal vector is interpolated and the intensity is computed by the illumination model equation shown in Eq. 1.

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_i f_{att_i} I_{\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L_i} + k_s O_{s\lambda} (\overline{R_i} \cdot \overline{V})^{Srm}], \quad (1)$$

where I_λ is the light intensity for one of three color indices, R,G or B, and k,O_d and O_s are reflection coefficient, diffuse color and specular color, respectively. N is the normal vector of the current vertex and $\overline{L_i}$ is the i-th light source's vector. $\overline{R_i}$ is the vector for the reflected light and $\overline{V}$ is the viewer's vector as shown in Fig. 1. These vectors are all normalized.

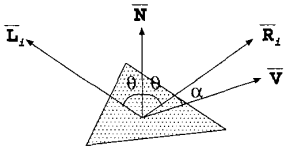


Fig. 1. Vectors used in the Phong illumination model.

In Eq. 1, the first term represents the ambient intensity which models the intrinsic intensity and the second term is the diffuse intensity which represents the brightness of the object. The third term is the specular color which exhibits the shininess of the object. The shininess is dependent on the angle, α, between the reflected light vector $\overline{R_i}$ and the viewer's vector $\overline{V}$, i.e, the inner product $\overline{R_i} \cdot \overline{V}$. The Phong illumination model assumes that the maximum specular reflectance occurs when the angle α is zero and falls off sharply as α increases. This rapid falloff is approximated by $(\overline{R_i} \cdot \overline{V})^{S_{rm}} = (cos\ \alpha)^{S_{rm}}$, where S_{rm} is the *specular reflection exponent* of a material. Phong introduced the specular exponent S_{rm} for the first time and this model has been accepted as a very good approximation of physical highlights.

Figure 2 shows the type of arithmetic operations required for implementing the Phong illumination. The cycle counts were

0-7803-5973-9/00/$10.00 ©2000 IEEE.

obtained by using a general DSP processor(TI's TMS320C67x).

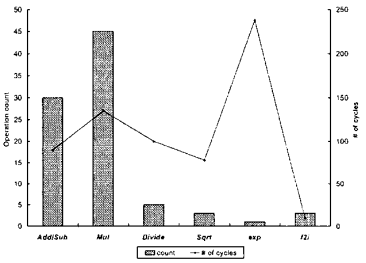


Fig. 2. Operation counts and cycle counts required for one computation of the Phong illumination on TI's TMS320C67x.

Although the operation count of exponentiation operations is relatively small compared to that of additions or multiplications, the cycle count of exponentiation operations is larger than that of additions or multiplications. In TMS320C67x, the cycle count used for the exponentiation operation is over a half of the total cycle count. The previous approaches for the computation of the specular term are explained in the next section.

III. Previous Methods for the Specular Term Computation

The specular term in the Phong illumination model is shown in Eq. 2, which requires the exponentiation operation of two floating-point numbers.

$$(\overline{R_i} \cdot \overline{V})^{Srm} = cos^{S_{rm}}\alpha = 2^{S_{rm} \times log(cos\ \alpha)} \quad (2)$$

Over the past years, there have been many works which attempted to compute the specular term fast and with small hardware. The exponentiation operation can be calculated directly by using the machine instructions provided that the exponentiation operation is supported in the machine as instructions or by using a software mathematical library. In AMD K6-II or Pentium, there are two instructions which calculate $log_2 x$ and 2^x where x is a floating-point number. With these instructions, $(cos\ \alpha)^{S_{rm}}$ can be computed as in Eq. 2 which takes about 150 cycles. Since it must be computed as many as *(number of points inside a polygon)* $\times$ *(number of light sources)* times per one polygon, the cycle count consumed for the operation is huge. It can be computed using a software algorithm which is based on normal integer and bit-operation instructions. For example, the $pow(a,b)$ function of a C library takes about 140 cycles.

Bishop and Weimer proposed a Taylor series approximation for the inner product of vectors and forward differencing of quadratic polynomials[9]. Deering proposed a normal vector shader which interpolates the normal and the eye vectors in hardware[4]. In both approaches, the exponentiation of a cosine value was done by the table lookup, which leads to an intolerable hardware size if a range of the exponent is large. To reduce the table size, a linear interpolation can be used, which requires 1 addition, 1 subtraction, 2 multiplications, 1 integer part extraction and 2 table accesses to compute an interpolation. All these operations are usually computed by floating-point operations. However, the linear interpolation often creates visible Mach bandings which can only be eliminated by taking larger tables or higher order interpolations, leading to a memory-speed tradeoff. For scenes with many different objects, the memory size is intolerable because the large table must be created for every object.

The specular term calculation can be approximated by its Taylor or Chebyshev approximation to replace the exponentiation function by a polynomial[11]. This technique works well when the specular reflection coefficient has small values. However, as the specular reflection coefficient increases, complex polynomials are needed to have high accuracy. In the Phong shading method using angular interpolation, the specular term is approximated by a piecewise quadratic function [12] that requires an angle parameter and an arc-cosine function which are very computation expensive.

This paper describes special hardware that computes the exponentiation operation in 4 cycles with small loss of accuracy. It can be easily implemented by expanding a conventional floating-point multiplier. It will be shown that the loss of accuracy caused by the proposed approximation algorithm is so small that human eyes cannot discern any differences between the image generated by the accurate exponentiation algorithm of SPARC's C library and the one generated by using the proposed approximation algorithm.

IV. Approximation of Exponentiation Operation

As in Eq. 2, $(cos\ \alpha)^{S_{rm}}$ can be computed by a sequence of the logarithmic and exponential operations. In this implementation, $cos\ \alpha$ is represented by the single precision floating-point format specified in IEEE 754[20]. IEEE 754 is the most widely used specification to represent floating-point numbers and to compute the floating-point operations. It has an 8-bit biased exponent and a 23-bit fractional part as shown in Fig. 3. Also, S_{rm} is stored in a special register as a fixed-point number.

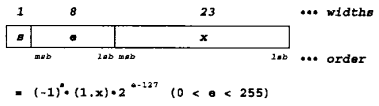


Fig. 3. Single precision floating-point number format of IEEE 754, where 'e' is a biased exponent with the bias of 127 and x is the fractional part of a mantissa.

As shown below, $log_2(cos\alpha)$ is approximated by a piecewise linear function. The approximation is developed to calculate the logarithm value with simple operations, that can be implemented with small hardware. A similar approximation was presented in [19], but our approximation is much easier to implement than that of [19].

$$\begin{aligned} & log_2(cos\ \alpha) \\ &= log_2(2^N(1+x))\ (N: exponent,\ 1+x: mantissa) \\ &\cong \begin{cases} N+1.25x & 0 \le x < 0.25 \\ N+x+0.0625 & 0.25 \le x < 0.75 \\ N+0.75x+0.25 & 0.75 \le x < 1.0 \end{cases} \end{aligned} \quad (3)$$

where N which is the same to "e-127" is the unbiased integer exponent of $cos\ \alpha$, and x is the fractional part that does not include the hidden bit. In other words, x represents the lower 23 bits with a decimal point at the 23rd bit in a single-precision floating-point number as shown in Fig. 4.

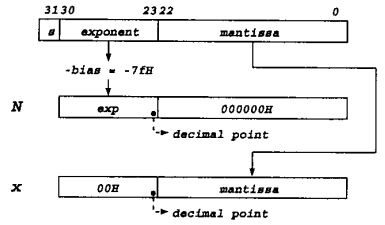


Fig. 4. "N" and "x" fields in $cos\ \alpha$ which is represented by a single precision floating-point number. N is an unbiased integer exponent and x is a fractional part.

The addends in Eq. 3 are easy to generate : $1.25x = x + (x >> 2)$ and $0.75x = x - (x >> 2)$. Moreover, it is also simple to determine whether $x < 0.25$, $0.25 \le x < 0.75$ or $x \ge 0.75$. If 2 MSB's of x are "11", x is greater than or equal to 0.75. If 2 MSB's of x are "10" or "01", x is between 0.25 and 0.75. Otherwise, x is less than 0.25. A circuit for finding the range of x is shown in Fig. 5. As an example, let us consider the case

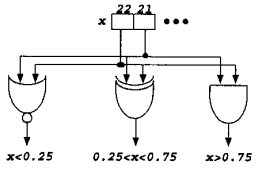


Fig. 5. The generation of three signals which determine the range of x.

of $x < 0.25$. Its approximated $log(cos\ \alpha)$ can be calculated as in Eq. 4, where "$A.B$" means that A is the upper 8-bit integer part and B is the lower 23-bit fractional part. Since $cos\ \alpha$ is in a range of 0 and 1 if the object is visible, its exponent is always smaller than the bias, and the sign of the approximated value is negative.

$$\begin{aligned} & N + 1.25x \\ &= N + x + (x >> 2) \\ &= -((bias.0 - (exponent.0)) - x - (x >> 2)) \\ &= -((bias.0 - ((exponent.0) + x)) - (x >> 2)) \\ &= -((bias.0 + (exponent.0 + x) + 0.000002\text{H}) \\ &\quad + (x >> 2) + 0.000002\text{H}) \\ &= -(\ (bias.0 + 0.000004\text{H}) + (exponent.0 + x) \\ &\quad + (x >> 2)\ ) \end{aligned} \tag{4}$$

The other cases when $0.25 \le x < 0.75$ and $x \ge 0.75$ can be formulated similarly, and the resulting equations for $log_2(cos\ \alpha)$ are summarized in Eq. 5, where 0.400000H and 0.100000H indicate 0.25 and 0.0625 of Eq. 3.

$$log_2(cos\ \alpha) \cong$$

$$\begin{cases} -((bias.0 + 0.000004\text{H}) + ((exponent.0) + x) \\ \quad + (x >> 2))\ \ for\ \ 0 \le x < 0.25 \\ -((bias.0 + 0.000002\text{H} - 0.100000\text{H}) \\ \quad + ((exponent.0) + x))\ \ for\ \ 0.25 \le x < 0.75 \\ -((bias.0 + 0.000002\text{H} - 0.400000\text{H}) \\ \quad + (exponent.0) + x) + (x >> 2)) \\ \quad for\ \ 0.75 \le x < 1.0 \end{cases} \tag{5}$$

When the *exponent* is 8-bit wide, the bias is 127 which is $7FH$ in a hexadecimal format. The $(exponent.0) + x$ is $cos\ \alpha$ because the *exponent* is the biased exponent of $cos\ \alpha$ and x is the mantissa part of $cos\ \alpha$ without the hidden bit. The equation for this case is shown in Eq. 6.

$$log_2(cos\ \alpha) \cong$$

$$\begin{cases} -(\ 7F.000004\text{H} + (\ cos\ \alpha\ ) + (x >> 2)\ ) \\ \quad for\ \ 0 \le x < 0.25 \\ -(\ 7E.F00002\text{H} + (\ cos\ \alpha\ )\ ) \\ \quad for\ \ 0.25 \le x < 0.75 \\ -(\ 7E.C00002\text{H} + (\ cos\ \alpha\ ) + (x >> 2)\ ) \\ \quad for\ \ 0.75 \le x < 1.0 \end{cases} \tag{6}$$

Three 32-bit additions are sufficient for calculating Eq. 6, which can be implemented using a CSA(carry save adder) and a 32-bit carry select adder. The CSA accepts three operands and generates a carry and a sum, and the final 32-bit adder adds the carry and the sum to generate a final 32-bit result. A hardware implementation for the log approximation unit is shown in Fig. 6, where the "const" means a constant in Eq. 5.

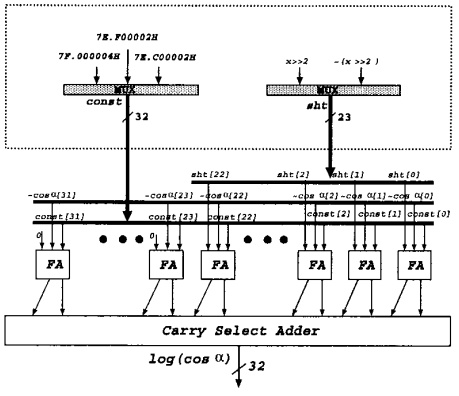


Fig. 6. Log approximation unit. A 32-bit CSA is used for the summation of three 32-bit numbers.

The output of the log approximation unit is a 32-bit fixed point number and it is shifted by 7 bits to make a 24-bit fixed point number. S_{rm} is saved in a special register as a fixed point number whose binary point is at the 16th bit. Upper 8-bit is an integer part and lower 16-bit is a fractional part. The approximated $log(cos\ \alpha)$ and S_{rm} is multiplied by the 24×24 multiplier in a floating-point multiplier unit. When this multiplier is used for the floating-point multiplication, it multiplies mantissas of two operands, but when used for log approximation it multiplies two fixed point numbers. The 24-bit multiplier is composed of 2 stages. The first stage is a CSA tree which has a

Booth encoded Wallace tree structure and the second is a carry select adder which adds the sum and the carry generated by the CSA tree. After the multiplication, the result is shifted by 9 bits to generate a 23-bit mantissa. If the integer part of the result exceeds the range of an 8-bit number, an overflow occurs, and the result is saturated to the maximum number which the result can represent.

The approximation equation for 2^x is shown in Eq. 7.

$$\begin{aligned} & 2^{S_{rm} \times log(cos\ \alpha)} \\ &= 2^{-(n+y)} \\ &= 2^{-(n+1)+(1-y)} \quad (n : integer, 0 < (1-y) < 1) \\ &\cong 2^{-(n+1)} \cdot 2^{\eta} \quad (0 < \eta (= 1-y) < 1) \end{aligned} \tag{7}$$

2^{η} is also approximated by piecewise linear equations. Because y is under 1, $\eta (= 1-y)$ is computed by inverting every bit of y. Eq. 8 is an approximation equation for 2^{η}.

$$2^{\eta} \cong \begin{cases} 1.0 + 0.75\eta & for\ 0 \le \eta < 0.25 \\ \eta + 0.9375 & for\ 0.25 \le \eta < 0.75 \\ 1.25\eta + 0.75 & for\ 0.75 \le \eta < 1.0 \end{cases} \tag{8}$$

$$= \begin{cases} \eta + (\eta >> 2) + 1.000002\text{H} \\ \qquad for\ 0 \le \eta < 0.25 \\ \eta + 0.F00000\text{H}\ for\ 0.25 \le \eta < 0.75 \\ \eta + (\eta >> 2) + 0.C00000\text{H} \\ \qquad for\ 0.75 \le \eta < 1.0 \end{cases} \tag{9}$$

$$= 1.0 + \lambda\ (\ 0 \le \lambda < 1.0\ ) \tag{10}$$

The coefficients used in the linear approximation are easily implemented by just shifting η and adding it with η. It is needed to convert $2^{S_{rm} \times log(cos\ \alpha)}$ to the floating-point number which has the biased exponent and the mantissa part. The mantissa has the hidden bit which does not appear on the representation but has the implied value of 1.0. Since all the approximated equations have the range of $1.0 < Approximated\ 2^{\eta} < 2.0$ as shown in Eq. 10, we just extract the lower 23 bits to get the fractional part of the final floating-point number. Because $n+1$ is an unbiased exponent, we have to add the bias to the integer part to obtain the exponent part.

Fig. 7 shows a hardware implementation of 2^n which is derived from Eq. 9, where n represents $S_{rm} \times log(cos\ \alpha)$. The constant for the $0 \le \eta \le 0.25$ case is not $1.000002H$ but $0.000002H$ because it is apparent that the MSB of $1.000002H$ will be the hidden bit.

Fig. 8 shows the data formats for $cos\ \alpha$ and S_{rm} when $cos\ \alpha = 0.45$ in decimal and $S_{rm} = 6.35$. As explained before, $log_2(cos\ \alpha)$ and S_{rm} are 24-bit wide and they are multiplied to generate $S_{rm} \times log_2(cos\ \alpha)$ which is 32-bit wide by shifting the 48-bit multiplication result.

Using the 24×24 multiplier in a floating-point multiplier unit, the circuitry for log approximation and exponential approximation can be merged into the floating-point multiplier. A block diagram of the merged FP multiplier called "Fastpow" unit is shown in Fig. 9. The saturation logic in the exponential approximation unit saturates the result when $cos\ \alpha$ is 1.0 or 0.0 and S_{rm} is 0.0. $(cos\ \alpha)^{S_{rm}}$ is 1.0 when S_{rm} is 0.0, and 0.0 when $cos\ \alpha$ is 0.0 and S_{rm} is not 0.0.

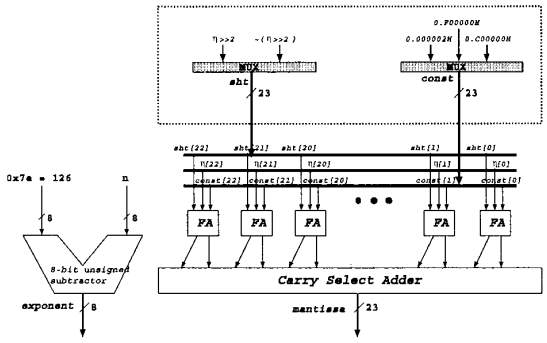


Fig. 7. Exponentiation approximation unit. An 8-bit subtracter is used to make the exponent part and a 23-bit CSA is used to make the fractional part.

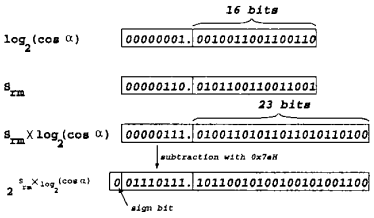


Fig. 8. Data formats of various values in "Fastpow".

As $cos\ \alpha$ increases, $(cos\ \alpha)^{S_{rm}}$ increases monotonically. If the proposed approximation equation doesn't monotonically increase as $cos\ \alpha$ increases, the image using this approximation equation may look significantly distorted. For instance, a brighter point in the original image can be seen as a darker point compared to the neighboring points. The monotonous increase of the approximation Eq. 3 and Eq. 8 used in "Fastpow" unit can be proved easily. Therefore, the image is almost the same as the original image if the error is not so large.

V. Results

The methods to calculate the specular term can be classified as follows.

- Table lookup method.

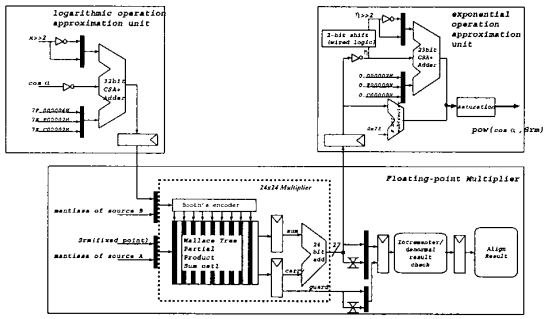


Fig. 9. "Fastpow" unit combined into a floating-point multiplier.

- Approximation algorithm(Taylor series, quadratic approximation).
- Software implementation of the exponentiation.
- Hardware internal instructions.

With a table lookup method, about 512 values of *cos* α are required and the differences of adjacent *cos* α values are saved in the tables for interpolation. Besides an additional memory to save the tables, the table must be updated whenever a new object appears. By quantizing both *cos* α and S_{rm} the table update can be removed, but a large sized ROM table is required. In this case, a $2^{12} \times 8bit$ ROM table is required usually. Also, the approximation algorithms such as Taylor series[11] or the quadratic approximation[12] requires multiplications and the calculation of arc-cosine values that take a lot of cycles. If internal instructions are used, about 160 cycles are used for both $log_2 x$ and 2^x instructions in Pentium or AMD's K6. If a software algorithm is used, about 140-200 cycles are required in the C libraries of various machines. Since such large overhead is inevitable in the specular term calculation, high speed applications that need many scene updates such as 3D games have not employed the specular reflection.

The proposed "Fastpow" algorithm takes just 4 cycles to compute the exponentiation of floating-point numbers. Therefore, the specular term calculation is very fast compared to the other methods. The hardware overhead is two CSA's, 32-bit adder, 23-bit adder and one 8-bit adder. The cycle count of "Fastpow" unit for the various operations is compared to other methods in Table I. We have assumed one floating-point ALU and one floating-point multiplier as basic hardware. The addition, subtraction, the conversion of floating-point number to integer is done in the floating-point ALU and the multiplication is done in the floating-point multiplier. We assume, as in Pentium, the latency of such an operation is 3 and that of a division is 39. The cycle count is roughly reduced to 1/10 of the others in "Fastpow" unit.

TABLE I

CYCLE COUNTS AND HARDWARE OVERHEAD FOR VARIOUS METHODS TO COMPUTE THE EXPONENTIATION FUNCTION.

	Cycles	Hardware overhead
Table lookup	$18 + 2\times$(table access)	Large ROM table
Taylor series	$\cong 68$	
Quadratic fn.	$\cong 44$	
Fastpow	4	CSA's, adders

The contour plot of absolute error is shown in Fig. 10. The x-axis is S_{rm} and y-axis is *cos* α. The maximum absolute error occurs when S_{rm} is small and *cos* α is close to 1.0. and the maximum error is 0.019217.

To inspect the effect of error on images, we have applied our algorithm instead of the *pow*() function to compute the specular term of the Phong shading. The images are compared in Fig. 11 for $S_{rm} = 10$ and Fig. 12 for $S_{rm} = 60$. Image (a) uses the specular term whose exponentiation value is calculated by the *pow*() function in Java language on a Pentium PC. Image (b) is generated by using the "Fastpow" approximation algorithm. As shown in the figures, image (a) and (b) containing 12261 Phong shaded pixels don't show any difference.

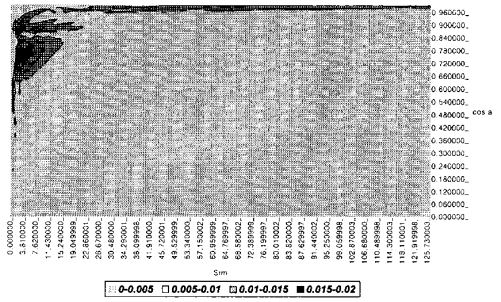
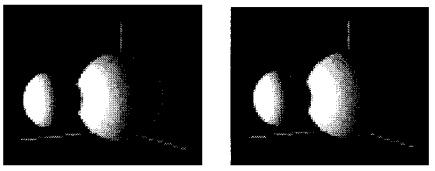


Fig. 10. The contour plot of absolute errors of "Fastpow". The absolute error was obtained by comparing with the value generated on a SPARC workstation.



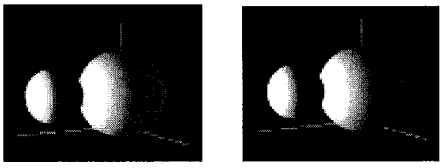
(a) With the specular term using Pentium's *pow*().

(b) With the specular term using "Fastpow".

Fig. 11. Two images when $S_{rm} = 10$.

The error of color values, Red, Green and Blue for each Phong shaded pixel are plotted in Fig. 13 that shows the color value difference between the image (a) and (b). Each color is represented by an 8-bit data and thus the maximum value is 255. The maximum error is 3 for Red, 3 for Green and 3 for Blue when S_{rm} is 10. When S_{rm} is 60, it is 2, 1 and 1. Error is very small compared to the maximum value 255.

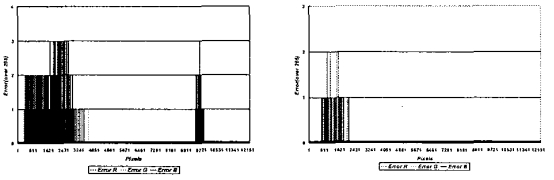
The PSNR(Peak signal-to-noise ratio) is a measure that is often cited as the quality of images. The calculation of PSNR defined in Eq. 11 is essentially the same as in the case of SNR except that in the nominator a hypothetical signal with a strength of the maximum value is used instead of the input



(a) With the specular term using Pentium's *pow*().

(b) With the specular term using "Fastpow".

Fig. 12. Two images when $S_{rm} = 60$.



(a) When $S_{rm} = 10$. (b) When $S_{rm} = 60$.

Fig. 13. Error of the ball image for each of R, G and B.

signal.

$$PSNR = 10\,log_{10}\frac{(maximum\ \ signal\ \ value)^2}{(mean\ \ square\ \ noise)} \qquad (11)$$

If we use PSNR as the measure of the algorithm we proposed, *maximum signal value* is 255 for 24-bit RGB and noise signal energy is the difference of a color value generated by using the *pow*() function and the other one from the "Fastpow". A large PSNR means that there is little difference between two images. The greater the PSNR, the closer the two images. The PSNR is different according to S_{rm}, but normally it is over 40dB for various values of S_{rm}. The PSNR of image (b) with respect to image (a) in Fig. 11 was 53.96dB, 61.27dB and 59.35dB for each R, G and B, and 68.05dB, 89.01dB and 75.59dB for Fig. 12.

VI. Conclusion

In this paper, we proposed the "Fastpow" unit which accelerates the calculation of the Phong illumination equation in 3D graphics. The "Fastpow"unit calculates the exponentiation value of two floating-point numbers in 4 cycles with small loss of accuracy while the exponentiation takes over 150 cycles or requires a large ROM table in other methods. The "Fastpow" unit can be merged into a conventional floating-point multiplier with ease and its hardware overhead is only two CSA's and three adders. We have also shown that the error is so small that there is almost no visual difference between two images generated by using the *pow*() function and the "Fastpow" unit.

References

[1] Bui-Tuong, Phong, *"Illumination for Computer Generated Pictures," Communications of ACM,* Vol. 18, No. 6, June 1975, pp. 311-317.

[2] Gouraud, Henri, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, Vol. 20, No. 6, 1971, pp. 623-628.

[3] Montrym, Baum, Dignam and Migdal, "InfiniteReality : A Real-Time Graphics System," *Proceedings of SIGGRAPH*, 1997, pp. 293-302.

[4] Deering, Michael, S. Winner, B.schediwy, C.Duffy and N.Hunt, *"The Triangle Processor Normal Vector Shader : A VLSI System for High Performance Graphics," Computer Graphics* , Vol. 22, No. 4, August 1988, pp. 21-30.

[5] Deering, Michael and S. Nelson, "Leo : A System for Cost Effective 3D Shaded Graphics," *Proceedings of SIGGRAPH*, 1993, pp. 101-108.

[6] Westin, Steven, Arvo, James and Torrance, Kenneth, "Predicting Reflectance Functions from Complex Surfaces," *Proceedings of SIGGRAPH*, 1992, pp. 255-264.

[7] Cook, Robert and Torrance, Kenneth, "A Reflection Model for Computer Graphics," *ACM Transactions on Graphics*, Vol. 1, No. 1, 1982, pp. 7-24.

[8] James D. Foley, Andries van Dam, Steven K.Feiner and John F.Hughes, *"Computer Graphics, Principles and Practise," Addison Wesley,* 1996.

[9] Bishop. G. and Weimer. D. M, "Fast Phong Shading," *Proceedings of SIGGRAPH*, 1986, pp. 255-262.

[10] Clausen. U. , "Reducing the Phong shading method," *Proceedings of Eurographics*, 1989, pp. 333-344.

[11] Pierre Poulin and Alain Fournier, "A Model for Anisotropic Reflection," *Proceedings of SIGGRAPH*, 1990, pp. 273-281.

[12] Kuijk. A. M and Blake. E. H, "Faster Phong Shading via Angular Interpolation," *Computer Graphics Forum*, 8, 1989, pp. 315-324.

[13] O. Nishii *et al., "A 200MHz 1.2W 1.4GFLOPS Microprocessor with Graphics Operation Unit," ISSCC Digest of Technical Papers*, Feb. 1997, pp. 402-403.

[14] Hiroshi Makino *et al., "A 286 MHz 64-b Floating Point Multiplier with Enhanced CG operation," IEEE J. Solid-State Circuits*, Apr. 1996, pp. 504-512

[15] J. Shipnes, *"Graphics processing with the 88110 RISC microprocessor,"* in *Dig. of Papers, IEEE Proc. COMPCON*, '92, pp. 169-174.

[16] J. Grimes, *"The Intel i860 64-bit Processor:A General-Purpose CPU with 3D Graphics Capabilities," IEEE Computer Graphics and Applications*, Vol. 9, No .4, July 1989, pp.85-94.

[17] J. H. Clark, *"The Geometry Engine: A VLSI Geometry System for Graphics," Computer Graphics*(Proc. Siggraph), Vol. 16, No. 3, July 1982, pp.127-133.

[18] Mark Segal and Kurt Akeley, "The OpenGL Graphics System : A Specification," Mar. 23, 1998.

[19] M. Combet, H. Van Zonneveld and L. Verbeek, *"Computation of the Base Two Logarithm of Binary Numbers," IEEE Trans. Electron. Compute.,* June 1975, pp. 863-867.

[20] *"Binary Floating-Point Arithmetic," IEEE Standards Board.,* March 1985.