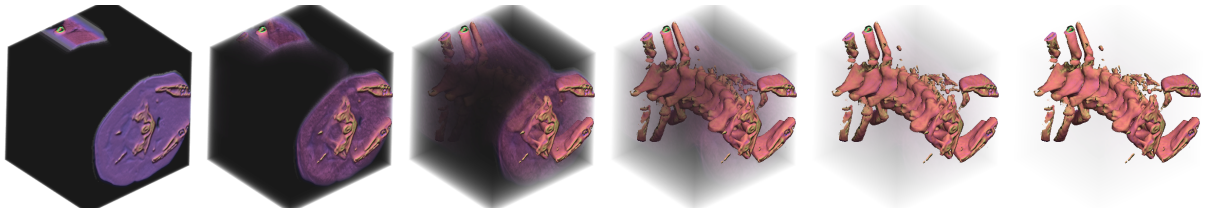
Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

Squeeze: Numerical-Precision-Optimized Volume Rendering

Ingmar Bitter
Clinical Image Processing Services
Department of Radiology
National Institutes of Health
IBitter@nih.gov

Neophytos Neophytou, Klaus Mueller, Arie E. Kaufman
Computer Science
Center for Visual Computing
State University of New York at Stony Brook
{nneophyt|mueller|ari}@cs.sunysb.edu



Abstract

This paper discusses how to squeeze volume rendering into as few bits per operation as possible while still retaining excellent image quality. For each of the typical volume rendering pipeline stages in texture map volume rendering, ray casting and splatting we provide a quantitative analysis of the theoretical and practical limits for the required bit precision for computation and storage. Applying this analysis to any volume rendering implementation can balance the internal precisions based on the desired final output precision and can result in significant speedups and reduced memory footprint.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation E.2 [Data]: Data Storage Representations

1. Introduction

Volume rendering has undergone a tremendous evolutionary process in the past decade and it has found a wide range of application domains in which it is routinely being used. The most obvious are the visualization of medical and scientific datasets, where the input data are most often inherently volumetric. However, we have also seen applications using volumetric sculpting to interactively generate volumetric data using volumetric deformation to modify the data on the fly. Many advances in software rendering algorithms [LL94, Lev88, MJC02, Wes90], datastructures [IL95, PSL*98, WV92], and custom hardware [MKW*02, PHK*99] have been made, and the recent revolution in commodity graphics hardware has helped volume rendering to gain speedups on the order of 1-2 magnitudes [EKE01, RSEB*00]. What has remained quite stable over the years is the volume rendering pipeline itself. Although we have seen a shift from pre-classified pipelines to post-classified ones (shown in Figure 1), this only requires a re-ordering of the pipeline stages to facilitate classification and shading after interpolation [MMC99]. Due to the massive amount of data that is involved in volume rendering, caching has received a fair bit of atten-

Figure 1: *Thin soft tissue using* $\alpha^{0.8b}$, $\sqrt{\alpha}^{0.8b}$, *and* $\alpha^{0.16b}$.

tion [Kni00, LCCK02, PSL*98]. All this resulted in a great deal of understanding on how to order the data items as they flow across the pipeline. However, what about the size of the data items? This is just as important because the size of the data items influences (i) the computational effort, (ii) the amount of data items that can be squeezed into the caches and registers, and (iii) the amount of data bandwidth that can be attained across both networks and buses. In this paper, we have attempted to address this important issue by analyzing the required data item precision at every stage of the volume rendering pipeline, and in this way to come up with the optimal data item size. The importance of maintaining sufficient precision was also observed in earlier work [MHB*00] when comparing several popular volume rendering approaches. In that research it was noticed that structures of low opacities could not be visualized well when using texture mapping hardware with limited bit-precision, especially when the preclassified pipeline was applied in conjunction with opacity-weighted color interpolation [WMG98].

In our discussion we have assumed that the precision is ultimately determined by the final stage, that is, the display on a computer monitor or projection wall, which is in most cases 8 bits per color channel. Although input and output media with higher precision do exist and our discussion readily generalizes to those, we feel that 8 bits will be a hard limit for a while because of the same limit within the DVI standard used by most LCD screens and the absence of any commodity displays with higher color resolution. We have also concentrated on two major volume rendering algorithms, raycasting and splatting. They can be implemented in four distinct ways. (i) First, one can write a software volume renderer that uses general purpose CPUs to compute the results of each raycast or splat volume rendering pipeline stage. (ii) Next, one can implement the same raycast or splat volume rendering pipeline stages using the same ray patters and the same math at 32bit precision using custom written shader programs for the new nVIDIA FX and ATI Radeon 9700/9800 chips. Hence, the rendering pipeline analysis is the same for this approach, only that any result exceeding 32bit precision can not be satisfied in the implementation. In addition, shader programs can use textures of varying bit depth and our analysis helps to decide which one to use. (iii) Third, one can use standard OpenGL or DirectX to implement texture map volume rendering that combines slices of 2D or 3D textures to form the volume rendered image. Mapping OpenGL or DirectX texture map functions to the splatting math is difficult and we are not aware of any published implementations. For raycasting a perfect match does not exist as well, but a significant subset is implementable. We point out where the general pipeline math can not be cast into standard OpenGL or DirectX calls. Texture map raycasting uses the same rendering pipeline stages, but is easier to implement with classification before interpolation. In order to accomodate this approach we analysed each pipeline stage seperately. The order inversion of these two stages actually have no effect on the outcome of the required precision. (iv) The forth approach to implementing raycasting or splatting is to build custom hardware. Here the sky is the limit, but hardware resources need to be carefully balanced to achieve maximum performance and our rendering pipeline analysis gives valuable pointers of how much precision to invest in each stage.

Our paper is structured as follows. First, Section 2 will discuss related work, while Section 3 will present some theory on fixed point arithmetic in the context of our special volume rendering scenario. Then, Section 4, will present our detailed precision analysis, addressing both theoretical and practical issues. Section 5 will present some results, and Section 6 will draw final conclusions.

2. Related Work

The VolumePro volume rendering chip [PHK*99] uses 12 bits throughout the pipeline, mainly motivated by the need to cater to the medical imaging industry that uses 12-bit data words. The Vizard II board [MKW*02] uses a 16-bit fixed point format in the compositing stage to achieve good image quality. In the color lookup tables, it allocates more bits (16) for opacities than for colors (8) to resolve low opacities when oversampling the rays. The PentiumIII based UltraVis system [Kni00] achieves high frame rates through extremely cache-friendly data layout and access. UltraVis uses 8-bit data as well as 16-bit fixed point color and transluceny buffers in the compositing stage. Finally, two major commodity graphics board manufacturers nVIDIA and ATI have recently made available 32-bit floating point arithmetic within their texture shaders, which brings plenty of precision, but requires non-standard vendor specific shader programs.

3. Fixed Point Arithmetic for Volume Rendering

In order to conduct our precision analysis within a common framework, we will be using fixed point notation. This is convenient since for volume rendering the range of useful results is generally contained in the tight interval of [0.0, 1.0]. This restricted interval has a number of important, as well as peculiar, consequences. Note that in any fixed point representation $I.Fb$ with I bits for the integer part and F bits for the fractional part the number 1.0 is represented as $I = 1$ and $F = 0$ or 1.0b. With this representation multiplying 1.0b with itself is achieved through multiplication of the integer variable with itself and a right shift by F bits, which results again in 1.0b. However, in OpenGL the range of α can be expressed as [0.0f,1.0f] as well as [0,255] which equates 1.0f with 255. If we use the [0,255] representation and try to compute 1^2 with one multiply and one shift we get $255^2 >> 8 = 254$. Due to truncation effects each successive multiplication with 255 will further reduce the re-

sult by 1. As we will see in the section about compositing 4.1 that we need to successively multiply total transparency (1.0). Hence, with the [0,255] representation of the unit range any attempt to composite complete transparency for 255 or more samples will counterintuetively result in total opacity. Consequently, we need more computational effort to use the [0,255] representation. One multiply, two adds and two shifts properly compute 1^2 [Bli95]:

$$\begin{aligned} tmp &= a^{[0,255]} * b^{[0,255]} + 128; \\ result &= (tmp + (tmp >> 8)) >> 8; \end{aligned} \quad (1)$$

However, a proper $I.Fb$ fixed point format can be used to compute the transparency compositing using only one multiply and one shift. With this approach the eliminated second shift and adds save half the computation time compared to the [0,255] representation. Therefore, the one saved bit in the [0,255] representation is not worth the computational cost, in particular in light of the fact that 8 bits are by far insufficient for the general case, as we shall see below. On a Pentium 4 both operations together actually take longer than a single floating point multiplication. Hence, for those CPUs it only pays to apply the fixed point math in practice if the need for multiplications is mixed with additions, in which case a single integer add can be used which is much faster than a floating point add.

4. Precision Analysis

Figure 2 illustrates the post-classification rendering pipeline stages for ray casting and splatting (specifically image-aligned sheet-buffered splatting [MSHC99]). The structure of both algorithms is very similar and so are most of their numerical precision requirements. Eventually both approaches result in an image that is usually displayed on a computer monitor. We focus here on mainstream PCs with 8 bits per color channel for the final display. These 8 bits per color channel are the crucial limitation for the achievable precision. We optimize all prior steps to have just enough precision to guarantee that the final image precision is not harmed. To propagate the limitations through the rendering pipeline, we analyze the pipeline stages in reverse order.

4.1. Compositing

The last step in the rendering pipeline is the compositing of the shaded samples yielding the final pixel value. For realistic-looking images $RGB\alpha$ tuples affecting the same pixel are interpreted as samples along a light ray into this pixel. Further, the light attenuation when traversing semi-transparent colored material is simulated [KV84, Kru91] using the *OVER* operator (see Equation 2) to ***composite*** consecutive $RGB\alpha$ tuples along the ray in sorted order. This blends the tuples with the proper partial occlusion. The computations have to be applied for each of the red, green and blue color channels, with any one of them denoted as C_λ.

$$\begin{aligned} {}^{OVER}\alpha C_\lambda &= C_{\lambda front} * \alpha_{front} + C_{\lambda back} * \alpha_{back} * (1 - \alpha_{front}) \\ {}^{OVER}\alpha &= 1 - (1 - \alpha_{front}) * (1 - \alpha_{back}) \end{aligned} \quad (2)$$

Compositing all samples contributing to the same pixel with the *OVER* operator in front to back order is called ***front to back compositing (F2B)***. It can be done through incremental compositing of color $C_{\lambda_{back}}$ and α_{back} into a compositing buffer (CB):

$$\begin{aligned} {}^{F2B}\alpha C_{\lambda CB} &+= C_{\lambda back} * \alpha_{CB} * (1 - \alpha_{CB}) \\ {}^{F2B}\alpha_{CB} &= 1 - (1 - \alpha_{CB}) * (1 - \alpha_{back}) \end{aligned} \quad (3)$$

(the += operator is borrowed from the C/C++ language). It is most efficient to incrementally composite α-pre-multiplied colors $\widehat{C}_\lambda = \alpha C_\lambda$ and transparency $T = 1 - \alpha$, thus, compositing $(\alpha R, \alpha G, \alpha B, T)$ tuples:

$$\begin{aligned} {}^{F2B}\widehat{C}_{\lambda CB} &+= \widehat{C}_{\lambda back} * T_{CB} \\ T_{CB} &*= T_{back} \end{aligned} \quad (4)$$

Blending with the *OVER* operator in the reverse direction is called ***back to front compositing (B2F)*** and results in the same final color:

$$\begin{aligned} {}^{B2F}\alpha C_{\lambda CB} &*= 1 - \alpha_{front};\ {}^{B2F}\alpha C_{\lambda CB} += C_{\lambda front} * \alpha_{front} \\ {}^{B2F}\alpha_{CB} &= 1 - (1 - \alpha_{CB}) * (1 - \alpha_{front}) \end{aligned} \quad (5)$$

Again, it is more efficient to composite $\widehat{C}_\lambda$ and T.

$$\begin{aligned} {}^{B2F}\widehat{C}_{\lambda CB} &*= T_{front} += \widehat{C}_{\lambda front} \\ T_{CB} &*= T_{front} \end{aligned} \quad (6)$$

Compositing all contributions along a ray yields the final transparency T_{CB}, which is a material property that is independent of the sampling rate. Hence, the transparency given

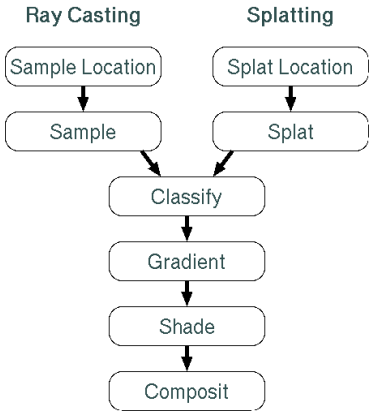


Figure 2: *The rendering pipeline stages for ray casting and splatting.*

to each sample has to be adjusted accordingly if the sampling rate is altered. For a homogeneous semi-transparent gel and a sampling rate of one sample per voxel and N samples along a ray, the final composited transparency becomes $T_{CB_N} = T^N$ (using Equation 6). If we now sample K times along the same ray, the sampling rate changes to $r = K/N$ and the new final transparency is $T_{CB_N} = \tilde{T}^K$. As the material has not changed, both expressions have to be equivalent:

$$\begin{aligned} \tilde{T}^K &= T^N \\ \tilde{T} &= T^{N/K} = T^r \\ \tilde{\alpha} &= 1-(1-\alpha)^r \end{aligned} \tag{7}$$

Adjusting α according to Equation 7 is called ***alpha correction*** [LCN98].

In a slice-order rendering algorithm, such as splatting or texture map volume rendering or Cube-4 style ray-casting [PK96], a complete slice of the intermediate results is stored in the compositing buffer which requires a significant amount of memory. A 512^2 $RGB\alpha$-image in floating point format uses 4MB, which is well beyond the on-chip cache sizes of special-purpose volume rendering chips. It is also well beyond the sizes of general purpose processors level-1 and level-2 caches. Hence, reducing the compositing buffer to the numerically required minimum will improve the cache hit ratio.

4.1.1. Theoretical Analysis

Multiplying two numbers in fixed-point format with f_1 and f_2 fractional bits each results in a new number with $f_1 + f_2$ fractional bits. During B2F and F2B compositing each sample requires a multiplication for color as well as for opacity or transparency. Thus, compositing of N samples requires Nf fractional bits for an absolute correct result. However, when constrained to a frame buffer with 8-bit precision, the compositing buffer precision may be reduced.

Compositing color $\widehat{C}_{\lambda CB}$ through Equation 4 or 6 accumulates the sample color contributions. With $N = 2^p$ samples along a ray, small contributions sum up such that the same low significance bit in each summand can influence a higher significance bit p positions further to the left in the final sum. The exponent p depends on the volume size and sampling rate, and it ranges from 8 for 256^3 volumes and one sample per voxel to 11 for $512^2 \times 2048$ volumes. Hence, to allow 8-bit precision of the final color, $8+p$ bits are needed to account for this accumulation of small contributions. Note that these small contributions may come from a saturated color being multiplied with a small opacity (or high transparency). Hence, opacity should be represented with with $8+p$ fractional bits as all of those can potentially affect the final composited result. Further, the final display of the rendered image is assumed to be on a 24-bit (R,G,B) display device which requires a conversion of $\widehat{C}_{\lambda CB} = \alpha C_{\lambda CB}$ to just $C_{\lambda CB}$. This conversion needs a division by α. If only the least significant bit of α is non-zero, then during the division the bits in $\alpha C_{\lambda CB}$ are shifted to the left by $8+p$ positions. Thus, over all, to allow 8-bit precision of the final color, $2(8+p)$ bits are needed to composite $\widehat{C}_{\lambda CB}$ which adds up to 32-38 bits.

Compositing using F2B (Equation 4) or B2F (Equation 6) both compute the transparency T through successive multiplications. If T is represented with the same precision as α using $t = 8+p$ fractional bits then T it is a number in $1.t$ format and is always in the value interval [0.0, 1.0]. Note, that during multiplication with any other number in the interval [0,1] none of the bits after the t^{th} have any effect on the first t fractional bits of the multiplication result. Hence, reducing the transparency compositing buffer precision to the same $8+p$ bits as each sample's transparency is sufficient which for our volume sizes and sampling rates equates to 16-19 bits. Note that this is only half for the precision needed for $\widehat{C}_{\lambda CB}$.

Compositing $RGB\alpha$ through Equation 3 or 5 adds more operations, but the number of multiplications and the number of sample contributions to sum up do not change. Hence, the precision analysis of $(\alpha R, \alpha G, \alpha B, T)$ also applies to $RGB\alpha$.

At a sampling rate r, applying alpha correction adjusts the transparency T to $\tilde{T} = T^r$. For sub-sampling $(r < 1)$ there are no precision limitations, but for super-sampling $(r > 1)$ the value of any $\tilde{T} \in (0,1)$ is less than that of T. The smallest change is needed for values close to zero and the smallest value of T at f fractional bits is 2^{-f}. Even for this value $0 < \tilde{T} < T$ still has to be true.

$$\begin{array}{rcrcrcl} 0 & < & \tilde{T} & = & T^r & < & T = T_{min} \\ 0 & < & \tilde{T} & = & \left(2^{-f}\right)^r & < & T = 2^{-f} \\ 0 & < & \tilde{T} & = & 2^{-fr} & < & T = 2^{-f} \end{array} \tag{8}$$

Equation 8 shows that the number of bits needed to represent $\tilde{T}$ is by a factor r larger than the bits needed for T. Hence, the assumed $8+p$ bits in the previous paragraphs need to be increased to $r(8+p)$.

The overall compositing buffer precision requirements are $r2(8+p)$ for $\widehat{C}_{\lambda CB}$ and $r(8+p)$ for T_{CB} and α_{CB}. This adds up to 1*2*(8+8) = 32 bits for T_{CB} and 1*(8+8) = 16 bits for α_{CB} for 256^3 volumes and no super-sampling. For $512^2 \times 2048$ volumes and 16 samples per voxel we get 16*2*(8+11) = 608 bits for T_{CB} and 304 bits for α_{CB}.

4.1.2. Practical Considerations

If we ignore the precision requirements due to alpha correction, we set r to 1 and just slightly modify the transparency of some structures without changing the overall impression. Fortunately, the largest error occurs at low transparencies, affecting samples that do not have much of a chance to accumulate the error. This is because low transparency is synonymous to most opaque and any contributions of samples behind a nearly opaque sample are usually not noticeable. In-

correctly snapping 99% opaque to 100% opaque is very acceptable. With this approximation the requirement for $\widehat{C}_{\lambda CB}$ becomes $2(8+p)$ bits and for T_{CB} or α_{CB} $8+p$ bits.

4.1.2.1. Opaque Iso-surface Visualization If texture map volume rendering is supposed to display only one or more iso-surfaces and if these are to be rendered as opaque surfaces, then the compositing buffer precision can be even more reduced. Storing the colors as $C_{\lambda CB}$ requires a division by α for each sample, but reduces the bit precision requirement to $8+p$ bits. Additionally, p can be reduced to 3, as usually no more than 8 non-zero samples are needed to reach opaque surface appearance. If we further assume that the colors assigned by the transfer function are very bright and can deal with a 3% inaccuracy ($\geq$8/256), then we can delete 3 more bits from the input color resulting in $5+p$ = $5+3$ = 8 bits for $C_{\lambda CB}$. *Hence, with opaque iso-surface oriented visualization configurations using 8 bits per channel are sufficient.* Note, that this restricted scenario allows texture map volume rendering to perform the compositing in the frame buffer with sufficient precision.

4.1.2.2. Fog Visualization Practical scenarios in which more bits are needed occur when an object is partially hidden in fog or bones are partially hidden behind nearly completely transparent soft tissue. Here, we cannot ignore the accumulation effect of small contributions building up over a long stretch of low opacity samples. If the volume is covered in fog and a structure whose iso-surface is supposed to be displayed is centered in the volume, each ray reaching this structure first traverses fog for about half the volume. This adds up to 1024 samples for a 512^3 volume and a sampling rate of 4 samples per voxel. In this scenario we have two special cases due to the high transparency values that need to be successively composited to T_{CB} according to Equation 4.

The first special case is concerned with the successively compositing of completely transparent material, and we already showed in Section 3 that we need to represent the transparency T as $1.Fb$ format number to maximize computation efficiency and ensure that $1^2 = 1$. The second special case is the soft tissue or fog that we do want to see, but only highly transparent such that the opaque structure or bone within is still fully visible. Using a fixed point format $1.7b$ with only 7 bits for the fractional part F, $\frac{2^7-1}{2^7} = \frac{127}{128}$ is the smallest partial transparency selectable. With this setup, each fog sample reduces the ray transparency by $\frac{1}{128}$, reaching full opacity after 63 samples. Therefore, the structure would be completely hidden by the fog. Increasing the fractional precision to 8, 10, 12, 14, 15, and 16 bits results in 1024 samples of thin fog accumulating to 100%, 100%, 25%, 6.25%, 3.08%, and 1.7% opacity, respectively. Note that here the color compositing buffer channels need at least the same precision as the transparency channel, but the per sample color does not need such high precision as it is used only once.

The teaser images on the title page illustrate the effect of the compositing buffer bit depth $b = 8, 10, 12, 14, 15, 16$ when rendering a human spine from a 512^3 dataset with a sampling rate of 2 and with all non-bone tissues and the background set to $\alpha = \frac{1}{2^b}$. This enables only the least significant bit and completely encloses the spine in this least-significant-bit-fog. It is clear that for this type of visualization 8 bits or 10 bits are not enough. 12 bits may be sufficient, but adjusting the fog density will only be possible in very noticeable discrete steps. 14 bits is sufficient for smooth fog adjustments. 15 bits would even allow increasing the sampling rate from 2 to 4 and still result in an image similar to the 14-bit fog image. *Hence, for general visualization settings, the compositing buffer should have 15 bits per color channel. A hardware implementation geared towards general volume visualization and* 512^3 *maximum volume dimension may reduce the compositing buffer to 14 bits per color channel.*

Placing the preferred fixed point bit precision 1.15b as "exponent", the preferred compositing equation is:.

$$\begin{aligned} {}^{F2B}\widehat{C}^{1.15b}_{\lambda CB} &\mathrel{+}= \widehat{C}^{1.15b}_{\lambda back} * T^{1.15b}_{CB} \\ T^{1.15b}_{CB} &\mathrel{*}= T^{1.15b}_{back} \end{aligned} \tag{9}$$

4.2. Shading

The color C_λ used in compositing results from evaluating the Phong illumination:

$$\begin{aligned} {}^{Phong}C_\lambda &= k_{ambient} O_\lambda I_\lambda \\ &+ k_{diffuse} O_\lambda \sum_{i=1}^{N_{lights}} I_{\lambda_i} \left(\vec{N} \bullet \vec{L}_i\right) \\ &+ k_{specular} \sum_{i=1}^{N_{lights}} I_{\lambda_i} \left(\vec{R}_V \bullet \vec{L}_i\right)^r \end{aligned} \tag{10}$$

For OpenGL and DirectX texture map volume rendering the specular term must be dropped because of the inability to compute the reflectance vector. However, it could be computed inside a custom shader program.

4.2.1. Theoretical Analysis

In Equation 10 the object color O_λ is usually an 8-bit value representing the range [0,1] read from a lookup table. Further, each I_{λ_i} is in the range [0,1] and the vectors are supposed to be normalized such that their dot product is in the range [-1,1]. However, a negative dot product signifies that the local surface element is facing away from the eye point such that its contribution should better be culled. Hence, all negative dot products are clamped to zero, which reduces the dot product result range to [0,1]. In addition, the sums $\sum_{i=1}^{N_{lights}}$ are clamped to the [0,1] range as well, and finally one can assume that $k_{ambient} + k_{diffuse} + k_{specular} = 1$ as those constants represent the weighting of the intensity contributions due to ambient, diffuse and specular light. Consequently, the overall sum will be in the range [0,1].

4.2.2. Practical Considerations

Knowing from Section 4.1.2 that the computed color needs 15 fractional bits and that the value range is [0,1], a 1.15b format to represent ${}^{Phong}C_\lambda$ is sufficient. However, each of the multiplications needs additional bits. These bits can either come from using standard floating point numbers or through the use of an extended fixed point format. If for each product one factor is in the range [0,1] and the other in the range [0,1), then a balanced fixed point format such as 16.16b will not overflow during the integer multiplication part. And if after each multiplication the result is shifted right by 16 bits all consecutive multiplications with more values in the [0,1] range do not overflow as well. If any of those factors is in the range (0,1) but still truncates the multiplication result to zero because of limited precision, we only incur a neglible error as the number of lights is usually small. Given eight lights the sum $\sum_{i=1}^{N_{lights}}$ could bring back eight contributions that are one bit below the least significant to the second to last position of the valid range. On the 16.16b format this is the difference between 0.003% intensity and 0% intensity, which is a very acceptable error. The only operation during the Phong equation evaluation that can not easily be replaced by integer math is the exponentiation needed for the specular light contribution. Knittel [Kni00] proposed to replace the exponential function with an inverted parabola that has a very similar shape and very similar area under the curve. Unfortunately, this approach has a discontinuous first derivative at the location where the contribution approaches zero. This discontinuity in the change of brightness is very visible to the human eye (see Figure 3). A better method is to restrict the exponent to powers of two and to compute the power through successive multiplications of intermediate results. If each multiplication truncates the results to 16 fractional bits then a value raised to the power of $2^6 = 64$ will have a maximum error of $1-(1-\frac{1}{2^{17}})^6 = 0.0046\%$ which results in much smoother perceived changes in brightness (see Figure 3). *Hence, for the use of fixed point math the Phong illumination equation can be evaluated using the 16.16b fixed point format:*

$$\begin{aligned} {}^{Phong}C_\lambda^{16.16b} &= k_{ambient}^{16.16b} O_\lambda^{0.8b} I_\lambda^{16.16b} \\ &+ k_{diffuse}^{16.16b} O_\lambda^{0.8b} \sum_{i=1}^{N_{lights}} I_{\lambda_i}^{16.16b} \left(\vec{N}^{16.16b} \bullet \vec{L}_i^{16.16b}\right) \\ &+ k_{specular}^{16.16b} \sum_{i=1}^{N_{lights}} I_{\lambda_i}^{16.16b} \left(\vec{R}_V^{16.16b} \bullet \vec{L}_i^{16.16b}\right)^{2^n} \quad (11) \end{aligned}$$

4.3. Gradients

Our analysis is based on the most commonly used gradient estimator, the Central Difference Gradient $G_{x,y,z}$, with $G_x = \frac{1}{2} sample_{(x+1,y,z)} - sample_{(x-1,y,z)}$, G_y and G_z analog.

4.3.1. Theoretical Analysis

The central difference gradient computation is straightforward. Given samples in a $I.Fb$ format the difference operator adds one bit of precision requirement to the integer part I and the division by 2 one bit to the precision requirement of the fractional part F. If the division by 2 is performed first, we do not even need the extra bit for the integer part. Hence, the result should be represented in a $I.(F+1)b$ format.

After normalization the gradient components are represented in 1.Fb format. This allows a discrete set of gradient directions with nearest neighbors having an angular difference of:

$$\sin\phi = \frac{1}{2^F}, \quad \sin\phi \approx \phi \;\Rightarrow\; \phi \approx \frac{1}{2^F}. \qquad (12)$$

This difference ϕ has the most severe influence in the specular lighting computation. With an exponent of 64 the power of the original dot product $1^{64} = 1$ may shrink to $(1-\frac{1}{2^F})^{64}$, which is a reduction by 22%, 6.1%, 1.6% and 0.4% if F is 8, 10, 12 or 14, respectively. Hence, for acceptable small differences in shading intensity, normalized gradients should

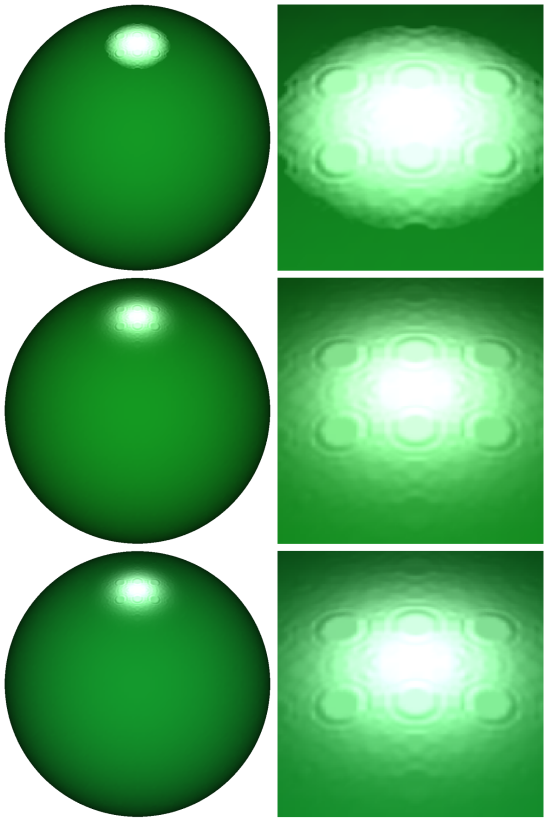


Figure 3: *A 512^3 sphere with Knittel's Phong approximation (top), with float precision and the pow() function of Phong (middle), and with fixed point Phong (bottom).*

be accurate up to at least 12 fractional bits. Figure 4 displays spheres rendered with a specular highlight and gradient precision of 4, 6, 8, 10, 12 and 14 bits. We observe that for 4-bit precision there are significant quantization artifacts in form of rings for the diffuse shading and an almost non-existent specular highlight. While 8-bit gradients are sufficient for the diffuse shading, the close-up images indicate that we need at least 12-bit gradients to obtain high-quality specular highlights.

4.3.2. Practical Considerations

For data from medical scanners the integer part of the samples often has 12 bits, while the fractional bits are set due to the sample interpolation computation. As shown in section 4.2 the normal used in the shading computation is expected to be normalized with components in the [0,1] range and in 1.15b format. If we perform this normalization the integer part of each component will be reduced from I to just 1 and there will be up to I bits added to the fractional part of each gradient vector component leaving only 4 interpolation bits to fill the remainder of the 16 significant fractional gradient vector bits. However, if the actual gradient magnitude is very small, then the normalization divides by a much smaller number and the fractional interpolation bits remain more significant. In the extreme all 16 bits can be influenced by the interpolation result.

In summary, both gradient computation and gradient normalization can be carried out using a 16.16b format.

This is a much harder requirement than can be satisfied by 8 bit per channel texture map implementations. Fortunately, this requirement is mostly based on the specular reflection contribution which texture map volume rendering must ignore anyway. The remaining diffuse shading is sufficiently represented with 8 bits such that iso surface oriented texture map volume rendering still can be achived without precision compromises.

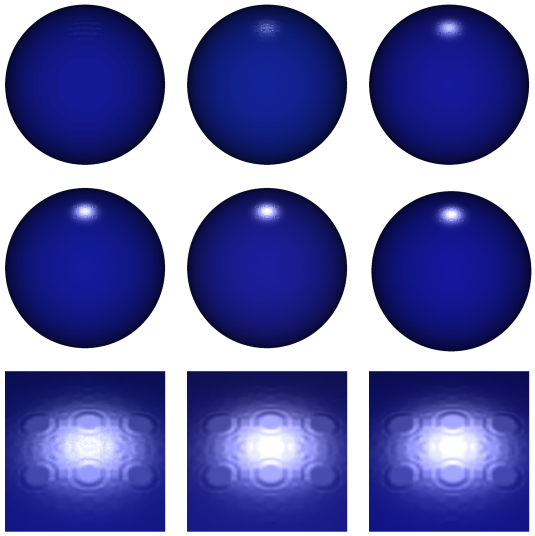


Figure 4: *First two rows:* 512^3 *spheres with gradient precision of 4, 6, 8, 10, 12, and 14 bits. Third row: close-ups for gradient precision of 10, 12, and 14 bits.*

4.4. Classification

Classification is the process of converting a voxel or sample into an $RGB\alpha$ tuple (the object color O_λ and the object opacity O_α). A ***color transfer function*** is used to map samples to colors. An ***opacity transfer function*** or ***α-transfer function*** is used to map samples to opacities α. It assigns high opacity to voxels and samples that should be visible and zero opacity to those that should be invisible.

4.4.1. Theoretical Analysis

As the transfer function may have high frequencies, the sampling rate used during rendering must be very high to avoid aliasing artifacts. The best way to achieve a high sampling rate without much computational cost is to use pre-integrated classification [EKE01, MGS02]. Beyond a data dependent sampling rate placing additional samples in between an existing pair of samples produces data values that are practically linearly distributed between the two corner values. Hence, one can restrict sampling during rendering to the corner samples and tabulate a very finely sampled integral for any pair of previous and current sample value.

4.4.2. Practical Considerations

With our focus on 12-bit input data a 1D transfer function table needs to have 4096 entries resulting in 32KB if 16 bits are used for each $RGB\alpha$ entry. The corresponding pre-integrated 2D table has 4096^2 entries, occupying 128MB. Because of this high storage requirement, precision is sometimes sacrificed and a 256^2 table of 8-bit $RGB\alpha$ is used instead, which occupies only 256KB. According to our previous analysis we have seen that we prefer to composite transparency and not opacity, and we have also seen that we need more bits in the transparency channel than in the color channels. Hence, a good compromise may be the $R^{0.8b}G^{0.8b}B^{0.8b}T^{1.15b}$ format. However, this results in 5 bytes per $RGB\alpha$ lookup table tuple, which is not very cache friendly. We have also shown that quantization errors for high transparency values have much larger effects than those for high opacities. Our preferred solution therefore uses only 8 bits, but distributes the values with increased concentration in the high transparency range. We achieve this by not storing T, but $\sqrt{\alpha}$ in the lookup table. Now, reading $\sqrt{\alpha}^{0.8b}$ from the table and then computing $T^{1.16b} = 1 - (\sqrt{\alpha})^2$ results in a 1.16b value for T. With this approach we use all 8 bits for the fractional part and, thus, doubling the number of possible values in the [0,1] range compared to using an

1.7b format for α. The reason to store a form of α and not T in the lookup table is that we can not distinguish between a 1.0f and $\frac{255^2}{256^2}$, which is a difference of 0.8%. This does not lead to perceivable image differences for these opacities, but would for high transparencies as shown earlier. Figure 1 depicts the differences in the thin fog scenario when using $\alpha^{0.8b}$, $\sqrt{\alpha}^{0.8b}$, and $\alpha^{0.16b}$.

Hence, the preferred lookup table entry has the format $R^{0.8b}G^{0.8b}B^{0.8b}\sqrt{\alpha}^{0.8b}$.

For texture map volume rendering this format is not an option and we have to resort to $R^{0.8b}G^{0.8b}B^{0.8b}0.8b$, which, again, provides sufficient precision as long as only iso-surfaces are being visualized.

4.5. Sample Interpolation / Splat Scan Conversion

To compute the value of a sample along a ray at an off-grid location, tri-linear interpolation is commonly used. Any tri-linear interpolation can be decomposed into seven linear interpolations. Similarly, when a splat is scan converted into a sheet buffer from its off grid splat center location, bilinear interpolation is used to find the splat weights at the pixel locations. Any bi-linear interpolation can be decomposed into three linear interpolations. Each linear interpolation involves four or three operations:

$$\begin{aligned} sample &= voxel_0 * (1-w) + voxel_1 * w && (13)\\ &= w * (voxel_1 - voxel_0) + voxel_0 && (14)\end{aligned}$$

4.5.1. Practical Considerations

The most precision we require of samples is for their use in gradient computation. We capped the gradient precision earlier to the 1.12b format after normalization. The second use of samples is as input for transfer function lookup in which case 12-bit integers are needed. As voxels are represented as 12.0b format the weights need to be in 1.12b format to ensure that in the extreme case the two voxel values that are being interpolated differ only by one, there are still 12 bits used in the dynamic range of the interpolated results. Thus we have enough precision for both possible scenarios: gradient estimation in low-dynamic range neighborhoods and in high-dynamic range neigborhoods. In the former case, normalization by a small gradient magnitude will shift the lower bits back to the left, while in the latter case we need the bits for the actual gradient computation. Hence, by providing 12.12b precision for the samples we are able to deal with both scenarios and therefore will be able to pick out even small density variations accurately (which makes most sense in datasets with low noise levels).

The preferred linear interpolation formula is therefore:

$$sample^{12.12b} = w^{1.12b} * (voxel_1^{12.0b} - voxel_0^{12.0b}) + voxel_0^{12.0b} \quad (15)$$

Texture map volume rendering is again restricted to 8 bits and only appropriate for iso-surface rendering.

4.6. Sample / Splat Location

To follow a ray through the data volume sample locations along the ray are computed at off grid locations. This computation is usually done incrementally. Hence, the k^{th} sample is at location $\vec{P}_0 + \sum \vec{V}_{inc}$. This involves k additions which can accumulate the error in the least significant bit of the $\vec{V}_{inc}$ and invalidate all of the last $\log k$ bits. Consequently, at four samples per voxel and a $512^2 \times 2048$ volume as many as 14 bits could be incorrect. The remaining bits of the location must be sufficient to represent an off-grid location that accurately interpolates the surrounding data values that uses 12 bits, as explained in the next section. And finally the non-fractional part has to have sufficient precision to enumerate all voxels, which requires up to 12 bits for $512^2 \times 2048$. Thus, theoretically, the only proper choice to compute ray sample locations is double precision.

4.6.1. Practical Considerations

In practice, the limited precision location representation in $I.Fb$ format has the effect that continuous rays are snapped to a discrete set of rays with an angular difference of

$$\sin\alpha = \frac{k\frac{1}{2^F}}{k}, \quad \sin\alpha \approx \alpha \;\Rightarrow\; \alpha \approx \frac{1}{2^F}. \qquad (16)$$

Hence, with only 6 fractional bits those discrete rays differ by 0.9 degrees, with 12 bits they differ by 0.05 degrees, and with 16 bits by only 0.0009 degrees. For parallel projection all rays in a single frame have the same direction alleviating any precision influence on image quality. During rotations a limitation to 0.9 degree steps is not perceivable so 6 bits precision are sufficient. For perspective projection 0.9 degree steps between neighboring rays are not enough, but 0.05 are, as this allows to distribute 1024 rays across a 60 degree field of view.

Further considering that we want to address locations in up to 2048 slices we need at least 11 bits for the integer part of the sample location. *Consequently, the minimum sample location precision is the 11.12b format.*

In Splatting, we deal with one "ray" per voxel instead of the one ray per image pixel during ray casting. The splat-ray traverses from the voxel to the eye. For parallel projection again all rays are parallel and again 6 fractional bits are sufficient to allow for smooth rotation. In the perspective projection case we can have volumes as large as $512^3 \times 2048$ with as many as 2048 voxels projecting from a line perpendicular to the view direction towards the eye. This is a factor of two more than the ray density during ray casting. *Therefore, to include splatting, we need one additional fractional bit resulting in a minimum sample location format requirement of 11.13b.*

For efficient splatting it is best to transform all contributing voxels first and store their transformed locations in sheet buffer bins according to their distance from the image plane. In addition, a u-index into the pre-integration splat table is needed that is derived from the transformed z-coordinate. After that the z-coordinate can be dropped and only x and y have to be stored to signify the pixel in the image and the weights for bilinear interpolation. Finally, the voxel value v is needed as well. Representing v using 8 bits all four values together can be stored as $x^{11.13b}u^{8.0b}y^{11.13b}v^{8.0b}$ which occupies a cache friendly total of 64 bits. After a table lookup only a single shift gets x or y into a processor register and a single masking AND to get u or v into a register.

5. Results

In summary, the minimum data precision requirements are as follows:

rendering stage	input	output
sample locations	n/a	11.12b
sample interpolation	12.00b	12.12b
classification	12.00b	0.15b
gradients	12.12b	1.12b
shading	1.12b	1.15b
compositing	1.15b	1.15b

For texture map volume rendering all data items have to be reduced to 8 bits and speculr reflections can not be computed, however, for the restricted case of visualizing one or possibly multiple iso-surfaces the OpenGL and DirtectX rendering pipelines do provide sufficient precision.

Note that the output data may be stored either as a single value or in an image-sized buffer. For raycasting, unless a ray-beam is traced (for tile-casting), all output items are computed on a pixel-basis and no buffers are needed. On the other hand, for the image-aligned sheet-buffered splatting algorithm both sample interpolation and compositing occurs on a sheet-basis. However, all other values can be computed on a pixel-basis. For rendering with slice-based texture mapping hardware a similar situation is given. Here, however, all data items have to be reduced to 8 bits and specular reflections can not be computed. Nevertheless, for the restricted case of visualizing one or possibly multiple iso-surfaces the OpenGL and DirtectX rendering pipelines do provide sufficient precision.

Modern PC CPUs are highly optimized to use integers that exactly fit the internal registers. A Pentium 4 can perform two simple 32-bit integer operations per clock cycle. Our analysis showed, that we never needed more than 16-bit precision for the integer part of our computations and also never more than 16 bits of precision for the fractional part. Hence, everything conveniently fits into a 16.16b fixed point format, which also happens to be the ideal fit considering hardware resources. We measured speedups of up to $4\times$ compared to using 32-bit floating point format.

It should be noted that fixed point arithmetic is not always faster than floating point arithmetic on today's CPUs. For example, while additions are faster for integer-based fixed point numbers, multiplications tend to be slower due the fact that integers use the floating point unit as well, but in addition incur an extra shift operation. At the current date, we do not know if the same holds true for the upcoming GPU and CPU generations. In practice, we use floating point arithmentic for all color channel operations, such as classification, gradients, shading, and compositing, while we use fixed-point arithmentic for all geometry operations, such as viewing transform, splat positioning, and rasterization. The floating point format will use 8 bits to store the exponent and one bit for the sign, which leaves 23 bits for precision in the narrow dynamic range volume rendering is using. Hence, both formats will suffice for our precision requirements, and we can simply pick the format that yields best performance in the CPU or GPU.

6. Conclusions

We have shown how the physical display hardware limitations of 8 bits per color channel propagate back through the rendering pipeline and how each pipeline stage can be optimized for its numerical precision requirements. The union of all rendering pipeline stage requirements is captured in the 32 floating point format. Double precision does not increase the perceived image quality. Alternatively, a 16.16b format can be used to achieve the same quality images. We also showed that for splatting transformed voxel locations are sufficiently stored in a 64-bit $x^{11.13b}u^{8.0b}y^{11.13b}v^{8.0b}$ format. For splatting and ray casting alike transfer function table entries are sufficiently stored in a 32-bit $R^{0.8b}G^{0.8b}B^{0.8b}\sqrt{\alpha}^{0.8b}$ format and partially composited pixels / fragments are sufficiently stored in a 64-bit $R^{1.15b}G^{1.15b}B^{1.15b}T^{1.15b}$ format.

7. Acknowledgements

This work was partially supported by Hewlett Packard Laboratories, by ONR grant N000140110034, NSF CAREER grant ACI-0093157, and DOE grant MO-068. We would also like to thank Tom Malzbender and Michael Meissner for many fruitful technical discussions.

References

[Bli95] BLINN J. F.: Jim Blinn's corner: Three wrongs make a right. *IEEE Computer Graphics and Applications 15*, 6 (Nov. 1995), 90–93.

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware* (2001), pp. 9–16.

[IL95] IHM I., LEE R.: On enhancing the speed of splatting with indexing. In *Proceedings of Visualization '95* (Oct. 1995), Nielson G. M., Silver D., (Eds.), IEEE, pp. 69–76.

[Kni00] KNITTEL G.: The ultravis system. In *Volume Visualization and Graphics Symposium* (Salt Lake City, Utah, Oct. 2000), IEEE, pp. 71–79.

[Kru91] KRUEGER W.: The application of transport theory to visualization of 3-d scalar data fields. *Computers in Physics 5*, 4 (July 1991), 397–406.

[KV84] KAJIYA J. T., VON HERZEN B. P.: Ray tracing volume densities. In *Computer Graphics, SIGGRAPH* (Minneapolis, Minnesota, July 1984), Christiansen H., (Ed.), vol. 18, pp. 165–174.

[LCCK02] LEVEN J., CORSO J., COHEN J., KUMAR S.: Interactive visualization of unstructured grids using hierarchical textures. In *Volume Visualization and Graphics Symposium* (Boston, MA, Oct. 2002), IEEE, pp. 37–44.

[LCN98] LICHTENBELT B., CRANE R., NAQVI S.: *Introduction to Volume Rendering*. Prentice Hall PTR, 1998.

[Lev88] LEVOY M.: Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8(5)* (May 1988), 29–37.

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Computer Graphics, SIGGRAPH* (Orlando, FL, July 1994), vol. 28, ACM, pp. 451–458.

[MGS02] MEISSNER M., GUTHE S., STRASSER W.: Interactive lighting models and pre-integration for volume rendering on PC graphics accelerators. In *Proc. Graphics Interface* (May 2002), pp. 209–218.

[MHB*00] MEISSNER M., HUANG J., BARTZ D., MUELLER K., CRAWFIS R.: A practical comparison of popular volume rendering algorithms. In *2000 Symposium on Volume Rendering* (2000), pp. 81–90.

[MJC02] MORA B., JESSEL J.-P., CAUBET R.: A new object-order raycasting system. In *IEEE Visualization 2002* (2002), pp. 203–210.

[MKW*02] MEISSNER M., KANUS U., WETEKAM G., HIRCHE J., EHLERT A., STRASSER W., DOGGETT M., FORTHMANN P., PROKSA R.: Vizardii: A reconfigurable interactive volume rendering system. In *2002 Eurographics/SIGGRAPH Workshop on Graphics Hardware* (2002).

[MMC99] MUELLER K., MÖLLER T., CRAWFIS R.: Splatting without the blur. In *IEEE Visualization '99 Conference Proceedings* (N.Y., Oct. 1999), Ebert D., Gross M.„ Hamann B., (Eds.), IEEE, ACM Press, pp. 363–370.

[MSHC99] MUELLER K., SHAREEF N., HUANG J., CRAWFIS R.: High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics 5*, 2 (Apr.June 1999), 116–134.

[PHK*99] PFISTER H., HARDENBERGH J., KNITTEL J., LAUER H., SEILER L.: The volumepro real-time ray-casting system. In *Computer Graphics, SIGGRAPH* (Los Angeles, CA, Aug. 1999), ACM Siggraph, pp. 251–260.

[PK96] PFISTER H., KAUFMAN A. E.: Cube-4 - a scalable architecture for real-time volume visualization. In *Symposium on Volume Visualization* (San Francisco, CA, Oct. 1996), ACM, pp. 47–54.

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98 Conference Proceedings* (Oct. 1998), Ebert D., Hagen H.„ Rushmeier H., (Eds.), IEEE.

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage-rasterization. In *2000 Siggraph/Eurographics Workshop on Graphics Hardware* (2000).

[Wes90] WESTOVER L.: Footprint evaluation for volume rendering. In *Computer Graphics, SIGGRAPH* (Dallas, TX, July 1990), vol. 24(4), ACM, pp. 367–376.

[WMG98] WITTENBRINK C. M., MALZBENDER T., GOSS M. E.: Opacity-weighted color interpolation for volume sampling. In *IEEE Symposium on Volume Visualization* (Chapel Hill, NC, Oct. 1998), IEEE, pp. 135–142.

[WV92] WILHELMS J., VANGELDER A.: Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (1992), 201–227.