Variable-Precision Rendering

Xuejun Hao
hao@cs.umd.edu

Amitabh Varshney
varshney@cs.umd.edu

Department of Computer Science and UMIACS
University of Maryland at College Park
College Park, MD 20742
http://www.cs.umd.edu/~{hao, varshney}

Abstract

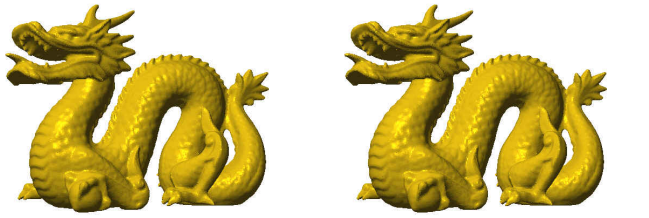
We propose the idea of using variable-precision geometry transformations and lighting to accelerate 3D graphics rendering. Multiresolution approaches reduce the *number* of primitives to be rendered; our approach complements the multiresolution techniques as it reduces the *precision* of each graphics primitive. Our method relates the minimum number of bits of accuracy required in the input data to achieve a desired accuracy in the display output. We achieve speedup by taking advantage of (a) SIMD parallelism for arithmetic operations, now increasingly common on modern processors, and (b) spatial-temporal coherence in frame-to-frame transformations and lighting. We show the results of our method on datasets from several application domains including laser-scanned, procedural, and mechanical CAD datasets.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Hierarchy and geometric transformations; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types.

Additional Keywords: hierarchical rendering, levels of detail, variable-precision rendering, view-dependent rendering.

1 Introduction

As the complexity of visualization datasets has increased beyond the interactive rendering capabilities of the graphics hardware, research in graphics acceleration has engendered several novel techniques that reconcile the conflicting goals of scene realism and interactivity. These techniques can be broadly classified into two lines of research. The first line of research includes techniques such as multiresolution rendering and visibility-based culling. Such techniques operate by reducing the number of graphics primitives to be rendered based on viewing and illumination parameters, such that there are minimal visually discernible differences between viewing higher complexity and lower complexity scenes. Orthogonal to these advances, we have been witnessing another line of research whose goal is to reduce the precision of each graphics primitive being rendered. Recently, reduction in precision of the object properties such as colors [11, 33], normals [7, 34], and vertex coordinates [16, 17] has been successfully attempted. The contribution of this paper lies in merging these two lines of research for variable-precision, view-dependent rendering.



(a) Floating Point (32 bits/vertex coordinate)

(b) Variable Precision (Average 7.6 bits/vertex coordinate)

Figure 1: Variable Precision Rendering

Most transformations and lighting for graphics primitives are currently carried out at full floating-point precision only to be later converted to fixed-point representation during the rasterization phase. An argument can be made that such high accuracy during geometry transformation and lighting stage sometimes exceeds even the display accuracy and thus causes several bits worth of unnecessary precision computation. We are currently witnessing four important trends in 3D graphics that have increased the need for variable-precision rendering:

1. View-dependent Rendering: View-dependent rendering has already introduced the concept of rendering different regions of a scene at varying geometric, illumination, and texture detail [14, 18, 32] based on their perceptual significance. A natural extension of this approach is to render each object at the precision appropriate for it. Under a perspective projection, objects that are close to the observer need more bits of precision than objects that are far.

2. Bounded Dynamic Range: Most graphics datasets have limited dynamic range. For instance, biomolecular datasets such as protein structures are determined using X-ray crystallography, NMR experiments, and gel electrophoresis. All of these methods have their accuracy limitations. Bounded input data accuracy with limited dynamic range also occurs in volumetric, range, and image-based datasets. Even in CAD datasets, the accuracy of the model is often limited by numeric round-off errors during intersection computations and precision limitations of acquisition methods which are greater than one in a million [21].

3. Processor-Level Support: With rapid growth in the size of the 3D datasets, geometry processing (transformation and lighting) has become a significant computational component of the 3D

graphics pipeline. To partly alleviate such computations in graphics and image processing, a variety of matrix math extensions to the CPU instruction sets have emerged: Intel's Pentium II with MMX and Pentium III with SSE, AMD's K6/Athlon with 3DNow!, and the Motorola PowerPC G4 with AltiVec. All of these instruction sets take advantage of SIMD (single-instruction multiple-data) parallel execution of instructions [12]. For instance, the Intel MMX [22] allows variable precision integer arithmetic to be implemented in SIMD parallelism where either two 32-bit, four 16-bit or eight 8-bit integer values are operated on in parallel. Such processor-level support for variable-precision arithmetic has enabled efficient implementation for variable-precision rendering.

4. Geometry Bandwidth Bottleneck: Increase in the geometric complexity of the graphics datasets has far outpaced the increase in the display complexity. This has resulted in a bottleneck in transferring 3D vertex data from the geometry processor to the graphics processor. More recently, graphics architectures such as the NVIDIA's GeForce 2^{TM} and S3's Savage 2000^{TM} , have emerged that perform transformation, lighting, and rasterization on the same chip. This has further shifted the bottleneck in the graphics pipeline from processing to bandwidth to and from the graphics chip. If the variable-precision rendering techniques discussed in this paper are adopted in a graphics API and/or implemented on the chip itself (in a manner similar to the MMX technology), this could significantly reduce the bus traffic to the graphics chip and accelerate the trasformation and lighting stages on the graphics chip beyond the results reported in this paper.

In this paper we lay down the mathematical groundwork for performing variable-precision geometry transformations and lighting for 3D graphics. In particular, we explore the relationship between the distance of a given sample from the viewpoint, its location in the view-frustum, to the required accuracy with which it needs to be transformed and lighted to yield a given screen-space error bound. The main contributions of this paper are:

1. We show how variable-precision transformations and lighting (at arbitrary precisions, not just 32 and 16 bits) can speed up general 3D transformations, parallel and perspective, and result in more efficient lighting.
2. We present a careful error analysis to relate the number of bits of input precision required for a given display accuracy.
3. We study how variable-precision operations can be used with spatial and temporal coherences.

2 Previous and Related Work

In computational geometry and solid modeling, research has been done on performing robust geometric operations. Exact rational arithmetic (i.e. in homogeneous coordinates) has been found to address several shortcomings of the conventional floating-point arithmetic [13]. However, successive geometric operations can result in an unbounded growth in the precision required to accurately compute the result. One way to limit the growth of the required precision is to intersperse rounding between arithmetic operations. Rounding-off vertex coordinates (or even line and plane coefficients [13, 28]) is reasonably well-understood now. However, such rounding is much more difficult if it must preserve some combinatorial or topological structure amongst the primitives (in/out, above/below, clockwise/counterclockwise orientation etc.). Several sophisticated approaches have been proposed that perform rounding and preserve some of these relationships by adding some extra points [8] or re-adjusting the rounded-off numbers to approximately maintain the relationships [19]. In a number of cases, such results are used only to establish topological relationships amongst primitives. This can be efficiently done by using sufficiently accurate (as opposed to exact) arithmetic [3, 9, 15].

Most of the research in graphics dealing with limiting the precision of vertex coordinates has focused on rounding-off the vertex coordinates (perhaps with attributes) independently of the topological structure defined by the vertices. Thus, with such approaches it is possible that the lower-precision models suffer from artifacts such as self-intersection and false incidences, even if the original higher-precision models did not. In practice, such artifacts have not been observed frequently enough yet, to convince most graphics practitioners to adopt the more time-consuming algorithms to preserve the topological structures. In this paper, we continue this line of thinking and quantize the vertex coordinates independently of the underlying topological constraints. Deering [7] has demonstrated that quantizing the normals down to 12 bits (i.e. only $4K$ unique normals) and vertex coordinates to 24 bits results in only minimal degradation in the rendered image quality. Reducing the precision of the vertex coordinates is implicit in the work of Rossignac and Borrel [27] and more recently, Luebke and Erikson [18]. The focus there is on reducing the geometric complexity of the high detail models. Consequently, even though the resulting vertex coordinates are effectively quantized on a grid and octree respectively, the reduced precision has not been taken advantage of during transformation and rendering.

Within the area of compression of 3D models, a lot of attention has been given to reducing the number of bits to represent vertex coordinates. Most approaches have used multi-stage quantization with Huffman encoding of delta-differences between successive vertices [2, 4, 7, 17, 29, 30]. Recently, progressive compression and transmission has been actively exploited [1, 6, 20, 21, 29]. Using the techniques of geometry prediction and progressive mesh encoding [1], combined with batch processing [6, 20] and entropy encoding [21], compression ratios for progressive compression have started approaching those for single resolution compression. King and Rossignac [16] have further balanced the reduction of the number of vertices and the reduction of bits per vertex coordinate using a shape complexity measure. For a nice survey of 3D geometry compression the interested reader may refer [2, 26]. Thus far, the reduced number of bits for representing vertex coordinates have not been used for speeding-up the rendering.

Researchers at Intel [22] have shown how to use the MMX^{TM} instruction-set to perform vector-matrix multiplies in short (16-bit) precision to speed-up the parallel projection. Specifically, they show how to perform four 16-bit operations in parallel.

3 Precision and Complexity

Let us first note the difference between multi-resolution and variable-precision rendering for 3D graphics models. Multi-resolution hierarchies have traditionally involved modeling each object at multiple levels of detail, where the detail is usually measured in the number of geometric primitives required for representation. Thus, a high-detail triangle-mesh object will require a higher number of vertices, edges, and triangles for representation. This complexity is largely independent of the precision at which each vertex is being represented. As can be seen in Figure 2(b), a multiresolution technique can be used to identify how *many* primitives are necessary for a faithful representation of a given object with a given set of viewing and lighting parameters. A variable-precision technique provides bounds on the bits of *accuracy* per primitive that are required for high-fidelity rendering. This can be seen in Figure 2(c) where the points selected to represent the circle all fall on the quantization grid. Thus, the two techniques are orthogonal to each other and depending on the application requirements for accuracy and speed can be used in a complementary manner.

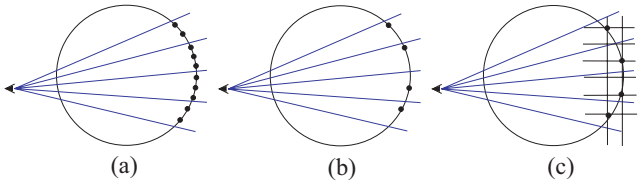


Figure 2: Varying Complexity versus Varying Precision

In the following analysis of different kinds of errors in geometric transformations, we shall assume that a minimum-sized cube has been constructed to cover the whole object using an algorithm similar to [10] and each axis has been normalized to the range of $[-1.0, 1.0]$. Thus the operands a and b are n-bit fixed-point representations of floating-point quantities within $[-1.0, 1.0]$. Additionally, we assume that we computed the n-bit fixed-point representation from such normalized floating-point representation by multiplying by 2^{n-1} and rounding to the nearest integer. Also, we would like to point out that this paper performs a worst-case analysis to guide the selection of appropriate precision. A good reference for sources and propagation of numerical errors is [24].

3.1 Representation Error

Often input data has uncertainty. A recent standards report from NIST outlines several types of uncertainty [31]. These include statistical (e.g., confidence intervals with mean and variance) and error (differences among estimates of the data from multiple sources and/or multiple time instants) uncertainties. Such uncertainties often limit the data acquisition precision. Other sources of error in the input data include approximations in the abstractions from which data is derived, numerical errors in computing the data using limited precision arithmetic, as well as instabilities in the mathematical models (as in ill-conditioned systems). Such errors often limit the number of bits of precision in the input dataset. For a n-bit fixed-point representation derived by rounding from a normalized floating-point representation, the representation error is at most half bit: $\varepsilon^{rep} \leq \frac{1}{2}$

3.2 Addition Error

For adding two n-bit integers, the error arises from the propagated error from the representation. $\varepsilon = \varepsilon^{gen} + \varepsilon^{prop} = \varepsilon^{prop} \leq \frac{1}{2} + \frac{1}{2} = 1$. So we will lose at most one bit of accuracy due to each addition.

3.3 Multiplication Error

We shall use $2n$ bits to store the intermediate result of multiplication of two n-bit integers. Since each normalized floating-point operand was magnified by a factor of 2^{n-1} during conversion to fixed-point before multiplication, we need to take out that extra 2^{n-1} factor by right shifting the intermediate result $n-1$ bits. The n-bit final result thus obtained has the largest error when both multiplier and multiplicand are close to 2^{n-1} and the absolute representation error is $\frac{1}{2}$: $\varepsilon \leq \frac{\frac{1}{2}\times 2^{n-1} + \frac{1}{2}\times 2^{n-1}}{2^{n-1}} = 1$. Thus, we lose one bit of accuracy due to each multiplication.

3.4 Division Error

Per-vertex division happens during the transformation from homogeneous coordinate to 3D image-space coordinates. The propagated error due to the division is:

$$\varepsilon^{prop} = \varepsilon_{\frac{a}{b}}{}^{prop} = \left|\frac{\partial(\frac{a}{b})}{\partial a}\right| \varepsilon_a + \left|\frac{\partial(\frac{a}{b})}{\partial b}\right| \varepsilon_b = \frac{\varepsilon_a}{b} + \frac{a}{b^2}\varepsilon_b$$

Here, ε_a and ε_b are the representation errors in a and b and are each at most $\frac{1}{2}$. For vertex within the view volume, we have $a \leq b$. Also, the generated error due to truncation is 1. Thus:

$$\varepsilon = \varepsilon^{gen} + \varepsilon^{prop} = 1 + \frac{\frac{1}{2}}{b} + \frac{a \times \frac{1}{2}}{(b)^2} \leq 1 + \frac{1}{b}$$

Since in viewing transformations, the divisor b is the distance of a scene vertex to eye in normalized view-volume representation (where the distance of the farthest point is 1.0),

$$\varepsilon \leq 1 + \frac{\text{distance of far plane in view-volume from eye}}{\text{distance of scene vertex from eye}}$$

So the loss of number of bits accuracy is

$$\left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}) \right\rceil$$

3.5 Putting it all together

For a 1024×1024 window, with pixel level accuracy, we need 10 bits in each x and y to represent the position of a vertex on the screen. Transformation of a vertex in homogeneous coordinates with a 4×4 matrix requires four multiplications and three additions for each coordinate. The height of this operation tree is three (leaves at level 3 have four multiplies, level 2 has two additions, and the root at level 1 has the final addition). Thus, we will lose 3 bits of accuracy in this matrix-vector multiplication. To get n bits of accuracy after transformation and homogeneous division, we need m bits to represent the input data:

$$m = n + 3 + \left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}) \right\rceil$$

Thus, if the display window is 1024×1024, $n = 10$ for pixel level accuracy; and if the distance of the point being rendered is half way across the view-volume, we shall need 15 bits to represent the vertex data: $m = 10 + 3 + \lceil \log_2(1+2) \rceil = 15$. This can be used to compute the requisite number of bits of precision required for each vertex based on its distance from the eye and forms the basis of view-dependent precision-based rendering.

For applications which require sub-pixel accuracy, we can increase the window resolution in the above formula. For example, if the application needs four bits of sub-pixel accuracy along each dimension, then we add four more bits to the requirements, which in the above example will result in a requirement of 19 bits of accuracy per input vertex coordinate for a 1024×1024 window.

4 View-dependent Transformation

The formula from the last section gives the upper bound on the number of bits needed to transform the vertices in order to get n bits of accuracy. In reality, if the object projects to the screen in an area that is small compared to the screen size, we may need less than n bits to get window-resolution-level accuracy. For view-dependent transformation, we have to find out the number of bits needed for vertices at different locations.

4.1 Octree-based Bounding Volume Hierarchy

To take advantage of the view-dependent information, we need an efficient way to estimate the projected size of different parts of an object. An octree bounding volume hierarchy is easy to build and very efficient to get the bounding volume of the projected vertices.

The idea is to find the minimum and maximum number of bits required for each bounding box using equations in the following subsections. If the two numbers are equal, then all vertices within this box will need the same number of bits during the transformation. Otherwise, vertices in this bounding box need different number of bits, and we should recurse to the lower levels of the octree hierarchy. In our implementation, we have used the normalized object coordinates, i.e., all x, y, and z coordinates lie within $[-1.0, +1.0]$.

4.2 Projected Size of the Dataset

For each view point, we calculate the projection of the eight corner points of the root level bounding box. From these projected points, we can find out the size of the object on the screen. The corner points are transformed into parallel-projection canonical viewing volume. The whole viewport will map into $[-1.0, +1.0]$ in both x and y direction of this viewing volume, so the relative size of the projected object to screen is just half the range of these projected corner points. During the calculation, we store the transformed minimum and maximum W value (i.e., minimum and maximum depth, W_{min} and W_{max}) of these eight points for later usage. The distance from the nearest visible scene vertex to view point is just the bigger one of W_{min} and near clipping plane.

4.3 Nearest Visible Vertex Accuracy

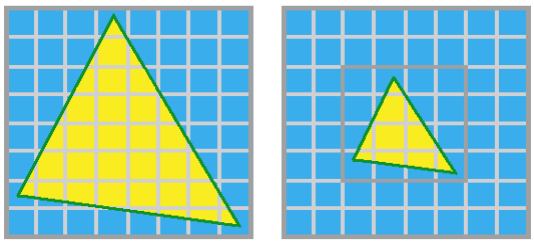
Given the width and height of the screen in number of pixels, the number of bits for pixel-level accuracy is (if sub-pixel accuracy is desired, just add those in):

$$n = screen\_bits = \max(\lceil \log_2 width \rceil, \lceil \log_2 height \rceil)$$

From last subsection, we know the ratio of the projected object size to the screen size is the projected range divided by 2, so the bits needed to represent the object will be $\left\lfloor \left| \log_2(\frac{\text{projected range}}{2}) \right| \right\rfloor$ bits less than $screen\_bits$.

Taking into account the computation error due to the multiplication, addition, and division as mentioned in the last section, the number of bits needed for the nearest visible scene vertex is:

$$near\_bits = n + 3 - \left\lfloor \left| \log_2(\frac{\text{projected range}}{2}) \right| \right\rfloor + \left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of nearest vertex to eye}}) \right\rceil$$



(a) Need 3 bits in x and y (b) Need 2 bits (and 1 bit of offset)

Figure 3: Object of Smaller Projected Size needs Less Precision

As shown in Figure 3, the smaller object in (b) only occupies less than half of the screen in each dimension, so it will need one bit less than the bigger object in (a). The extra screen offset will be added in the final viewport transformation step.

4.4 Accuracy to Represent Each Vertex

Due to the perspective foreshortening, an object appears smaller as its distance to the viewpoint increases. As an example, an object at twice the distance will have half the size on the screen, and thus needs one less bit to represent.

Generally, given the $near\_bits$ as defined before, we try to find the number of bits for any other vertex. After the transformed W value (i.e., depth) is known, we calculate the vertex_bits as:

$$vertex\_bits = near\_bits - \left\lfloor \log_2(\frac{\text{transformed } W \text{ of this vertex}}{\text{distance of nearest vertex to eye}}) \right\rfloor$$

It will be expensive if we need to do this calculation for each vertex. Fortunately, with the bounding box hierarchy, very few calculations need to be done.

Starting from the top of the hierarchy, we calculate the minimum and maximum transformed W value for that node. First we calculate the transformed W value of the center of the node (denoted as W'_{center}), then we check the eight corner points of the node to figure out the minimum and the maximum. As we already have the W_{min} and W_{max} of the corner points at the root level, for the subtree at level k:

$$W'_{\min} = W'_{center} + \frac{W_{\min}}{2^k} \quad \text{and} \quad W'_{\max} = W'_{center} + \frac{W_{\max}}{2^k}$$

where the denominator is due to the fact that the node size is reduced by a factor of two when we go down one level in the octree.

Using the above two equations, we can find out the minimum and the maximum number of bits needed for vertices within the box:

$$V_{min} = near\_bits - \left\lfloor \log_2(\frac{W'_{\max}}{\text{distance of nearest vertex to eye}}) \right\rfloor$$

$$V_{max} = near\_bits - \left\lfloor \log_2(\frac{W'_{\min}}{\text{distance of nearest vertex to eye}}) \right\rfloor$$

If these two numbers are equal, then we know that all the vertices within the box will need these number of bits to represent. Otherwise, vertices in the box require different numbers of bits and we need to recurse down one more level of the octree.

5 Spatio-Temporal Coherence

In the last two sections we have seen the relationship between the input bits of accuracy and the bits of accuracy required for the output. For the same number of bits of accuracy for the output, we can further reduce the bits of accuracy required in the input by taking advantage of spatial and temporal coherence. This can result in further savings in processing time as well as in the bandwidth to the graphics processor.

5.1 Spatial Coherence

The basic idea that we use to take advantage of the spatial coherence is that the difference in spatially close vertices can usually be represented in far fewer bits than those required to represent each vertex coordinate in its entirety. This idea has been used with great success in the research on 3D compression of geometry as discussed in

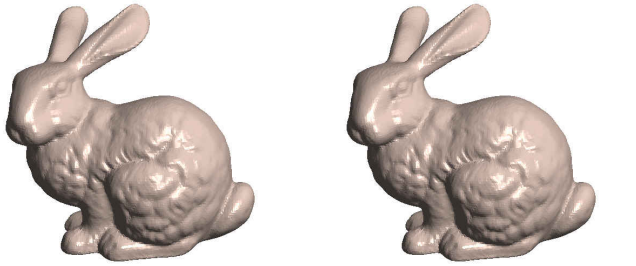
Section 2. If a vertex coordinate x' can be represented by a delta-difference with respect to another coordinate x as $x' = x + \Delta x$ then one can decompose the transformation for coordinate x' as: $Mx' = M(x + \Delta x) = Mx + M\Delta x$

Since the number of bits of accuracy required to transform Δx is much smaller, one can perform several of them in parallel. To exploit this idea, we can partition the dataset by any spatial subdivision scheme such as an octree over the vertices of the model. In our implementation we have used an octree that subdivides by the volume centroid at each level. In this scheme, since each level reduces the range by half, the vertices in each lower level require one bit less than their parents. The accuracy of the transformation of a vertex coordinate with a matrix is represented by the lower of the two accuracies. Thus if the vertex coordinates can be quantized in less bits, the transformation matrix values can also be quantized with less number of bits.

In this approach we independently transform the delta difference in the vertex coordinate position between the current level of the octree and its parent. Then we can get the final transformed results for each vertex by a top-down tree traversal as the following: ($LIMIT$ is the lowest level of tree below which the difference between the transformed parent and children is negligible)

```
Top-Down-Tree-Traversal(x)
if x ≠ NULL
    if x.level ≤ LIMIT
        x.value = x.parent.value + x.transform
        for i from 1 to 8
            Top-Down-Tree-Traversal(x.child(i))
    else
        x.value = x.parent.value
        for i from 1 to 8
            Top-Down-Tree-Traversal(x.child(i))
```

As an example, if we could operate on byte and short precision operands and we required 16 bits of accuracy, then we could transform the top eight levels of the octree in short-precision and the lower levels could be transformed in byte-precision (or even lesser, if available). By using such hierarchical schemes, one can get a better precision efficiency without losing accuracy. Figure 4 compares the results of bunny model using floating point and variable-precision transformation.



(a) Floating Point Transform (32 bits/vertex coordinate)

(b) Variable Precision Transform (Average 7.9 bits/vertex coordinate)

Figure 4: Variable-Precision Transformation of Bunny Model (Stanford model, 69K triangles; lighting for both images has been calculated in floating point)

5.2 Temporal Coherence

Similar to the idea of spatial coherence, we can take advantage of temporal coherence by noting that the difference in the transformed vertex positions does not differ significantly from one frame to the next. Thus if we calculate the difference in the transformation matrix from one frame to the next and use the difference matrix ΔM to transform a vertex, we can then add it to the previously transformed vertex position in less number of bits: $M'x = (M + \Delta M)x = Mx + \Delta Mx$. Extending this idea further, we note that one can combine the spatial and temporal coherences: $M'x' = (M + \Delta M)(x + \Delta x) = Mx + \Delta Mx + M\Delta x + \Delta M\Delta x$ As we show in Tables 2 and 3 for the Auxiliary Machine Room dataset, the average number of bits that are operated upon for each vertex as well as the equivalent number of operations can both be greatly reduced by taking advantage of both spatial and temporal coherences.

6 Variable-Precision Lighting

Color is usually represented by 8-bits of precision in red, green, and blue components and sometimes even less (for instance in lower-range graphics cards and Personal Digital Assistants). Also, if depth cueing is turned on and the far objects are displayed at lower intensities, their color can be represented using fewer bits.

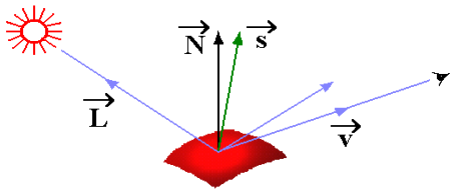


Figure 5: Lighting Calculation

Before we go to the detailed treatment of the variable-precision lighting, let us review the formula for the lighting calculation we have used. Although there are good psychophysically-based light reflection models [23], we decided to implement the OpenGL illumination model. As in OpenGL, we assumed diffuse and Phong illumination with Gouraud shading without per-pixel normal evaluation:

$$\begin{aligned} Color \;=\; & \text{emission}_{\text{material}} + \text{ambient}_{\text{light\_model}} * \text{ambient}_{\text{material}} + \\ & + \sum_{i=0}^{m-1} \left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i * (\text{spotlight effect})_i * \\ & (C_{\text{ambient}} + C_{\text{diffuse}} + C_{\text{specular}})_i \end{aligned}$$

where m is the number of light sources, $(\frac{1}{k_c + k_l d + k_q d^2})$ gives the attenuation factor in which d is the distance between the vertex and the local light source. $C_{\text{ambient}} = \text{ambient}_{\text{light}} * \text{ambient}_{\text{material}}$, $C_{\text{diffuse}} = (\max\{\overrightarrow{L} \cdot \overrightarrow{N}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}}$ and $C_{\text{specular}} = (\max\{\overrightarrow{s} \cdot \overrightarrow{N}, 0\})^{\text{shin}} * \text{specular}_{\text{light}} * \text{specular}_{\text{material}}$. $\overrightarrow{L}$ is the unit vector that points from the vertex to the light position, $\overrightarrow{N}$ is the unit normal vector at the vertex, $\overrightarrow{s}$ is the normalized half way vector between the directions of the light source and the viewer, and *shin* is the shininess, i.e., the specular exponent. Our goal here is to find the necessary number of bits to represent the input illumination data in order to get the required accuracy in output color.

6.1 Sources of Error in Local Illumination

There are several additional sources of error in local illumination computation beyond the sources of error we have already discussed in the transformation stage (representation error, addition error, multiplication error, and division error). In lighting computations we have to deal with addition and multiplication errors for operands with different bits of accuracy, the square root operation error which results from vector normalization, and the error induced by exponentiation in specular illumination. Also, the special case of dot product of two unit vectors is worthy of separate analysis.

To reduce the error propagation, we can multiply the light coefficient with the object material property coefficient in floating-point form before converting to the n bits fixed-point representation. For example, instead of converting $ambient_{\text{light\_model}}$ and $ambient_{\text{material}}$ to n bits of integer, we multiply them in floating-point representation and then convert the result to n bits of integer. This way, we can save one bit of accuracy which will be lost due to the multiplication of two n-bit integers. We next consider the other sources of error in the following subsections.

6.1.1 Error for Operands with Different Accuracy

Let us consider two operands with different bits of accuracy, say n and n' where $n' < n$. This means that if the maximum possible value is 1, then the representation errors are $2^{-(n+1)}$ and $2^{-(n'+1)}$, respectively. For addition, the error ε can be computed as:

$$\varepsilon \leq 2^{-(n+1)} + 2^{-(n'+1)} = 2^{-(n'+1)}(1 + 2^{-(n-n')})$$

As an example, if $n - n' = 2$, then: $\varepsilon \leq 2^{-(n'+1)}(1 + \frac{1}{4})$. The error will stay at $(n'+1)^{th}$ bit, and the result will get n' bits of accuracy, i.e., the same accuracy as the less accurate operand.

Similarly, for multiplication of operands with n and n' $(n' < n)$ accuracy, the maximum possible error happens when the operands are close to the maximum possible value which we treat as 1, as we discussed in last section:

$$\varepsilon \leq 2^{-(n+1)} * 1 + 2^{-(n'+1)} * 1 = 2^{-(n'+1)}(1 + 2^{-(n-n')})$$

Again, the result has the same accuracy as the less accurate operand.

6.1.2 Error in the Dot Product of Unit Vectors

Let us consider two unit vectors, say $\overrightarrow{\alpha}$ and $\overrightarrow{\beta}$, with n bits of accuracy in each of their three components:

$$\overrightarrow{\alpha} = (\alpha_1, \alpha_2, \alpha_3) \text{ and } \overrightarrow{\beta} = (\beta_1, \beta_2, \beta_3)$$

Since the error in the three components ε_{α_i} and ε_{β_i} $(i = 1, 2, 3)$ is in the $(n+1)^{th}$ bit, i.e., $2^{-(n+1)}$, their dot product error is:

$$\varepsilon_{(\overrightarrow{\alpha} \cdot \overrightarrow{\beta})} = \sum_{i=1}^{3}(\beta_i \varepsilon_{\alpha_i} + \alpha_i \varepsilon_{\beta_i}) \leq 2^{-(n+1)}(\sum_{i=1}^{3}\alpha_i + \sum_{i=1}^{3}\beta_i)$$

For unit vector $\overrightarrow{\alpha}$, we have: $\alpha_1^2 + \alpha_2^2 + \alpha_3^2 = 1$.
From the inequality: $a^2 + b^2 \geq 2ab$, we get:

$$(\alpha_1 + \alpha_2 + \alpha_3)^2 \leq 3(\alpha_1^2 + \alpha_2^2 + \alpha_3^2) = 3$$

So we have $(\alpha_1 + \alpha_2 + \alpha_3) \leq \sqrt{3}$. Similarly, we have $(\beta_1 + \beta_2 + \beta_3) \leq \sqrt{3}$. Then:

$$\varepsilon_{(\overrightarrow{\alpha} \cdot \overrightarrow{\beta})} \leq 2^{-(n+1)}(\sum_{i=1}^{3}\alpha_i + \sum_{i=1}^{3}\beta_i) \leq \sqrt{3} * 2^{-n}$$

That means, we will lose one to two bits of accuracy for dot product of two unit vectors.

6.1.3 Error in the Square Root Operation

For lighting calculations we need to normalize the vectors to unit length before we compute the dot product. Normalization involves division by the magnitude of the vector which requires a square root operation. In order to perform all the operations in the fixed-point arithmetic, we use a a table lookup to get the square root of an unsigned integer.

For an unsigned integer X with $2n$ bits of accuracy we take the most significant n bits (say X') as the lookup index into the square-root table to find the square root.

$$X' = (X >> n) << n$$

The maximum possible error of X' relative to X is 2^n (because the information in the lower n bits is lost). We can reduce this error by half though. If the value of the n^{th} bit of X is one, we can add one to $X >> n$, so that it becomes a kind of rounding error instead of trunction error.

Next, we use the square-root table to find the square-root of X'. Let this be a' in integer representation: $X' = a'^2$. Suppose the square root of X in integer representation is $a : X = a^2$. Let $a' = a + \varepsilon_a$ (ε_a is the error), then:

$$a'^2 = (a + \varepsilon_a)^2 = a^2 + 2a\varepsilon_a + (\varepsilon_a)^2 = X'$$

That is, $2a\varepsilon_a + (\varepsilon_a)^2 = X' - X \leq 2^{n-1}$.

If $X > 2^{2n-2}$, then $a > 2^{n-1}$, thus:

$$2a\varepsilon_a < 2a\varepsilon_a + (\varepsilon_a)^2 \leq 2^{n-1} \text{ and } \varepsilon_a < \frac{2^{n-1}}{2a} < \frac{2^{n-1}}{2*2^{n-1}} < \frac{1}{2}$$

Which means that if we use the most significant n bits of the unsigned integer as index into the square root table then as long as the integer is bigger than 2^{2n-2}, the result has n bits of accuracy.

6.1.4 Error in the Evaluation of Specular Exponentiation

To calculate the specular component of illumination, we have to compute the exponent of the dot product of half-way vector with the normal vector. Due to the fact that the dot product a of two unit vectors is always smaller or equal to 1 and that we are only dealing with positive values of the dot product, we use 2^m to represent the largest value 1. The maximum possible representation error will be $\frac{1}{2}$, i.e., $2^{-(m+1)}$ relative to 1.

If ε_a is the error in the value a of the dot product, then:

$$(a + \varepsilon_a)^n \cong a^n + na\varepsilon_a \text{ (if } \varepsilon_a << a)$$

The maximum absolute error happens when $a = 1$, $\varepsilon_a = 2^{-(m+1)}$, and n is the maximum value of 128: (as implemented by OpenGL)

$$na\varepsilon_a < 128 * 2^{-(m+1)} < 2^{-(m-6)}$$

So we will have $m - 6$ bits accuracy in the result, i.e., we will lose 6 bits accuracy due to this exponentiation.

6.1.5 Putting it all together

From the above analysis, we can get an equation which relates the input data accuracy with the output color accuracy. Assume the output color needs n bits accuracy per R, G, and B, which requires m bits of accuracy in the input data. We next relate n and m.

First, the normalization of each vector will lose 1 bit. As shown before, the square root will have nearly the same accuracy as the input data. To avoid the loss of accuracy due to division, instead of storing the square root, we store the reciprocal of the square root in the lookup table. This reciprocal is calculated in the floating-point

representation before converting it to the n bits fixed-point representation. Thus the only error induced in the normalization will be in the final multiplication which is a loss of one bit of accuracy.

The dot product of two unit vectors will lose one to two bits of accuracy. Since the exponentiation will lose six bits, the term $(\max\{\vec{s}\cdot\vec{N},0\})^{\text{shin}}$ will lose $1+(1\text{ to }2)+6=8$ to 9 bits of accuracy. So the above term have between $m-8$ and $m-9$ bits of accuracy. Further, the term C_{specular} will have the same accuracy because $\text{specular}_{\text{light}} * \text{specular}_{\text{material}}$ will have m bits of accuracy, which is much higher than the accuracy of $(\max\{\vec{s}\cdot\vec{N},0\})^{\text{shin}}$.

Similarly, the term C_{diffuse} will get between $m-2$ and $m-3$ bits of accuracy. And we know the term C_{ambient} will have m bits of accuracy as explained in the overview.

Overall, $(C_{\text{ambient}}+C_{\text{diffuse}}+C_{\text{specular}})$ will have the accuracy as the lowest accurate term C_{specular}, i.e., $m-8$ or $m-9$ bits of accuracy.

Since the attenuation and the spotlight terms can all be evaluated with more than $m-8$ bits of accuracy, the required color accuracy bits of the entire illumination equation can be expressed as:

$$n = m-8 \text{ or } m-9$$

For example, if $n=8$, i.e., eight bits per R, G, and B, then the required accuracy for the input data will be $n+8$ or $n+9$, i.e., we will need 16 or 17 bits to represent the input data to get the desired accuracy of 8 bits per color component.

6.2 View-dependent Variable-Precision Lighting

Similar to the case of transformation in Section 5, we can take advantage of the spatial coherence of the adjacent vertices in lighting calculations.The basic idea is that the viewing and lighting directions do not vary much for the spatially close vertices. Once we find those directions for one vertex, we can calculate the directions for the nearby vertices incrementally, i.e., calculate the difference in far fewer number of bits. The direction difference between the nearby vertices depend not only on the absolute spatial difference of the vertices, but also on their distances from the viewer and light source to the vertices. Once the viewer moves closer to the vertex and below a threshold (which we will describe below) we will switch back to the original case, i.e., treat that particular vertex independently of its adjacent vertices.

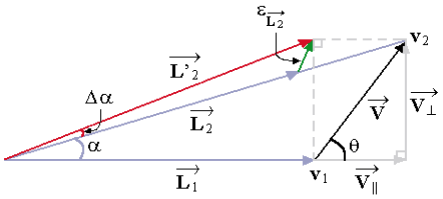


Figure 6: Incremental Lighting Calculation

In Figure 6 we show how to compute the lighting incrementally. Let $\overrightarrow{L_1}$ be the light vector for vertex v_1 for which we have already calculated the illumination. Now we would like to find out the light vector $\overrightarrow{L_2}$ for its adjacent vertex v_2. The displacement vector $\overrightarrow{V}$ between v_1 and v_2 is normalized by the distance between the v_1 and the light source, i.e., its length is equal to the real distance between v_1 and v_2 divided by the distance between the v_1 and the light source.[1] Both $\overrightarrow{L_1}$ and $\overrightarrow{L_2}$ are unit vectors.

One way to accurately compute $\overrightarrow{L_2}$ is to normalize the sum of $\overrightarrow{L_1}$ and the vector between v_1 and v_2. This approach requires roughly the same amount of computation as to compute the $\overrightarrow{L_2}$ directly from the vector between v_2 and the light source. To reduce the computation, we instead use $\overrightarrow{L'_2}$ (equal to $(\overrightarrow{L_1}+\overrightarrow{V_\perp})$) as the approximation of $\overrightarrow{L_2}$ if it satisfies our accuracy requirements. $\overrightarrow{V_\perp}$ is the component vector of $\overrightarrow{V}$ on the perpendicular direction of $\overrightarrow{L_1}$, which can be easily computed: $\overrightarrow{V_\perp}=\overrightarrow{V}-\overrightarrow{L_1}(\overrightarrow{V}\cdot\overrightarrow{L_1})$. Using this approach, the induced error $\varepsilon_{\overrightarrow{L_2}}$ is equal to $\overrightarrow{L'_2}-\overrightarrow{L_2}$.

If the length of $\overrightarrow{V}$, $\left\|\overrightarrow{V}\right\|$, is much smaller than 1 (the length of $\overrightarrow{L_1}$ and $\overrightarrow{L_2}$), then we have:

$$\begin{aligned}\left\|\overrightarrow{L'_2}\right\| &= \sqrt{\left\|\overrightarrow{L_1}\right\|^2+\left\|\overrightarrow{V_\perp}\right\|^2}=\sqrt{1+\left\|\overrightarrow{V_\perp}\right\|^2}\\ &\approx 1+\frac{\left\|\overrightarrow{V_\perp}\right\|^2}{2}\le 1+\frac{\left\|\overrightarrow{V}\right\|^2}{2}\end{aligned}$$

Let the angle between $\overrightarrow{L_1}$ and $\overrightarrow{L_2}$ be α, between $\overrightarrow{L_2}$ and $\overrightarrow{L'_2}$ be $\Delta\alpha$, and $\overrightarrow{V_\parallel}$ be the component vector of $\overrightarrow{V}$ along the direction of $\overrightarrow{L_1}$:

$$\alpha=\arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{\left\|\overrightarrow{L_1}\right\|+\left\|\overrightarrow{V_\parallel}\right\|}\right)=\arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{1+\left\|\overrightarrow{V_\parallel}\right\|}\right)$$

$$(\alpha+\Delta\alpha)=\arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{\left\|\overrightarrow{L_1}\right\|}\right)=\arctan\left(\left\|\overrightarrow{V_\perp}\right\|\right)$$

$$\begin{aligned}\text{So}\qquad \Delta\alpha &= \arctan\left(\left\|\overrightarrow{V_\perp}\right\|\right)-\arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{1+\left\|\overrightarrow{V_\parallel}\right\|}\right)\\ &\approx \left\|\overrightarrow{V_\perp}\right\|\left(1-\frac{1}{1+\left\|\overrightarrow{V_\parallel}\right\|}\right)=\frac{\left\|\overrightarrow{V_\perp}\right\|\left\|\overrightarrow{V_\parallel}\right\|}{1+\left\|\overrightarrow{V_\parallel}\right\|}\\ &< \left\|\overrightarrow{V_\perp}\right\|\left\|\overrightarrow{V_\parallel}\right\|\le\frac{1}{2}\left\|\overrightarrow{V}\right\|^2\end{aligned}$$

The last inequality is because $\left\|\overrightarrow{V_\perp}\right\|=\left\|\overrightarrow{V}\right\|\sin\theta$ and $\left\|\overrightarrow{V_\parallel}\right\|=\left\|\overrightarrow{V}\right\|\cos\theta$, and $\sin\theta\cos\theta=\frac{1}{2}\sin(2\theta)\le\frac{1}{2}$. Thus if the distance between v_1 and v_2 is much less than the distance between v_1 and the light source, then $\left\|\overrightarrow{L'_2}\right\|\approx 1=\left\|\overrightarrow{L_2}\right\|$ and $\Delta\alpha << 1$, therefore

$$\left\|\varepsilon_{\overrightarrow{L_2}}\right\|\approx 2\left\|\overrightarrow{L_2}\right\|\tan\left(\frac{\Delta\alpha}{2}\right)\approx\Delta\alpha\le\frac{1}{2}\left\|\overrightarrow{V}\right\|^2$$

This means, the error of using $\overrightarrow{L'_2}$ as an approximation of $\overrightarrow{L_2}$ is less than $\frac{1}{2}\left\|\overrightarrow{V}\right\|^2$. If we want 15 bits of accuracy in $\overrightarrow{L_2}$, we only need $\left\|\overrightarrow{V}\right\|\le 2^{-7}$, i.e., the distance between v_1 to the light source should be $2^7=128$ times larger than the distance between v_1 and its addjacent vertex v_2. This way we only use the local spatial differences in calculating the new direction and avoid an expensive vector normalization operation.

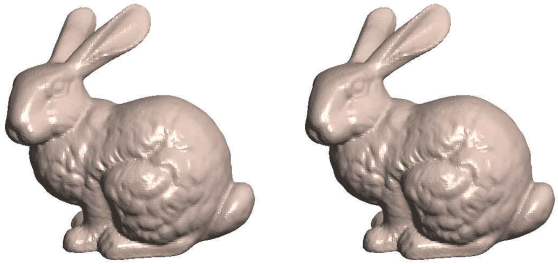
7 Some Implementation Details

In addition to what we have already described in the previous sections, there are some other implementation details which are worth mentioning.

[1] Note that this assumption is not shown in Figure 6, where $\overrightarrow{V}$ is shown to have its length as the distance between v_1 and v_2.

Model		Bunny	DHFR	Dragon	Venus	AMR	Buddha
Size (triangles)		69K	145K	202K	268K	376K	1087K
Floating Point	Transform	61	130	185	230	330	968
	Lighting	469	1042	1374	1830	2503	7481
	Other	56	108	167	218	298	902
	Total	586	1280	1726	2278	3131	9351
Variable Precision	Transform	17	33	46	59	83	235
	Lighting	79	155	212	280	337	882
	Other	42	87	127	160	212	616
	Total	138	275	385	499	632	1733
Speedup		4.25	4.65	4.48	4.57	4.96	5.40
e_{rms} (object space)		1.3e-4	1.3e-4	1.2e-4	1.2e-4	1.1e-4	1.2e-4
Max error (obj. space)		3.0e-4	3.1e-4	3.0e-4	2.9e-4	2.6e-4	3.1e-4
e_{rms} (image space)		8.5e-3	8.8e-3	8.7e-3	6.0e-3	8.4e-3	7.0e-3

Table 1: Results from Rendering at Varying Precisions



(a) Floating Point Lighting (b) Variable-Precision Lighting (Speedup: 2.99)

Figure 7: Variable-Precision Lighting of Bunny Model
(Transformations have been calculated in floating-point)

7.1 Batched Transformation and Lighting

Most graphics APIs (OpenGL, Direct3D, Glide) allow the user to transform and light the triangles one at a time and send the transformed and lighted triangles in floating-point screen coordinates to the rasterizer. Since these APIs do not accept screen-space triangles in the fixed-point representation, we had to convert our fixed-point results to floating-point representation before asking the graphics API to rasterize the triangles. In MMX technology, this means that we need to reset the register flag back and forth when we switch from the integer operation to floating point because these two share the same registers. The frequent resetting costs time, so the intuitive solution is to minimize the number of resets, e.g., transform and light the whole dataset first in object space, then do the viewport transform and then send to the rasterizer. There are two problems with this approach. First we lose some opportunities of pipelining which the hardware is very smart at. Second, there are lots of extra memory accesses due to the write-back, so this does not work well.

To solve this problem, we make a tradeoff. Instead of transforming and lighting the triangle one by one or all at the same time, we do them batch by batch. The resetting of the flag only happens between batches and we avoid the extra memory accesses. In practice, we find batch size of several hundred triangles works gracefully. If the graphics APIs accepted screen-space fixed-point representation triangles, we would not have to deal with this and our results would have been better than reported here since switching from fixed-point to floating-point is expensive even when we do them in batches.

7.2 Full-precision Matrix Calculation

At each view point we first calculate the transformation matrix and then apply it to all the vertices in the dataset. The initial matrix calculation is a negligible fraction of the overall computation which includes transformation of hundreds of thousands of vertices. So we compute it in full precision floating point before converting it into the fixed-point representation. This way, we save the precision of the matrix elements, and avoid the possibility of error build up when we take advantage of the temporal coherence of the frames in transformation because the matrix is computed in full precision separately for each frame.

8 Results

We have tested our approach on polygonal datasets from several application domains including laser-scanned, mechanical CAD, and procedurally generated datasets. The results of our approach are summarized in Tables 1, 2, 3 and appear in Figures 1, 4, 7–14.

We obtained the results shown in the paper and the video on a Pentium II 400MHz PC with 128MB RAM and a Voodoo3 3500 graphics card. Table 1 compares the results using variable precision with the one using traditional single-precision floating point and times are reported in milliseconds. The variable precision rendering showing here is under the requirements of guaranteed pixel-level position accuracy and eight bits per R, G, and B color. The object space root-mean-square error and maximum error are measured in transformed object space as the distance between the single precision floating point transformed vertices and variable precision transformed ones, while the image space root-mean-square error is measured in the final image space as the difference between the R, G, B color components. The formula for image space root-mean-square error is the following:

$$e_{rms} = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\widehat{f}(x,y) - f(x,y) \right]^2 \right]^{1/2}$$

Where $f(x,y)$ represents the original image, $\widehat{f}(x,y)$ denotes an estimate of the image, and $M \times N$ is the image size.

From Table 1, we can see that under the pixel-level accuracy, the maximum transformed distance between the two methods is less than 0.00033 for all the six datasets tested . We know the normalized transformed object space is in the range [-1.0, +1.0], so the difference is less than six-thousandth of the total range. This shows robustness of our method. Further, instead of getting pixel-level accuracy, our method actually gave us 2 to 3 sub-pixel bits of accuracy. This is because our error analysis gives the upper bounds of the error; the real error is usually much less. To roughly compare how variable precision rendering stacks up against multiresolution rendering, we compared the object space Hausdorff error in a 16K triangle model of the Bunny using Metro [5] against a 69K triangle model of the Bunny using 7.9 bits/vertex coordinate. Although both give a factor of 4 speedup, the variable precision method has an order of magnitude smaller object space Hausdorff error (0.012% of the bounding box diagonal) compared with 16K triangle full precision model (0.12% of the bounding box diagonal).

We can see more than a factor of four speedup in all the datasets tested. One aspect of our algorithm is that it scales well. The speedup factor goes up with the increase of scene complexity (which means more data will be rendered in less precision) and the

Output bits	Conventional		Spatial		Spatio-Temporal	
	Add	Mult	Add	Mult	Add	Mult
32 bits	32	32	42.21	31.55	54.08	29.23
16 bits	16	16	12.95	7.77	13.77	4.02
8 bits	8	8	1.35	0.66	0.77	0.08
4 bits	4	4	0.03	0.02	0.01	0.001

Table 2: Average Number of Bits per Vertex Coordinate Operated upon for Appropriate Output Precision

Output bits	Conventional		Spatial		Spatio-Temporal	
	Add	Mult	Add	Mult	Add	Mult
32 bits	6	8	7.92	7.89	10.14	7.31
16 bits	3	4	2.43	1.94	2.58	1.01
8 bits	1.5	2	0.25	0.17	0.14	0.02
4 bits	0.75	1	0.01	0.004	0.002	0.0002

Table 3: Average number of equivalent 32-bit operations per vertex coordinate for appropriate output precision

number of light sources. See Figure 8 and Table 1.

Figure 9 shows the histogram of percentage of vertices transformed in different number of bits for AMR dataset, which has a very low average 4.18 bits/vertex coordinate for variable-precision transformation, instead of 32 bits/vertex coordinate as in the single-precision floating point case. Figures 1, 10–14 show the images rendered by variable precision rendering and compare them with the single-precision floating point rendering. Even from the closeup there is hardly any visual distinguishable difference.

Table 2 shows the average number of bits that have to be manipulated per vertex during the transformations while exploiting spatial and temporal coherences. Since the vertices that are at the lower levels of the octree require less number of bits for transformation, the overall average number of bits turns out to be much less. The leftmost column indicates the number of bits that are required in the output display.

Table 3 shows the average number of equivalent 32-bit operations per vertex during the transformations while exploiting spatial and temporal coherences. Central to this idea is that one 32-bit operation is equivalent to two 16-bit, four 8-bit, and eight 4-bit operations. Even though SIMD parallelism at the level of 4-bits is not yet available in the current generation of processors, the table shows the effectiveness of our scheme if such parallelisms were to become available in future. As in table 2, the leftmost column indicates the number of bits that are required in the output display.

9 Conclusions

We have presented a novel approach to take advantage of SIMD parallelism in modern processors to speedup the transformation and lighting stages of the graphics pipeline. Our approach can successfully trade-off the precision for speed without significantly affecting the visual quality of the rendered images. In addition, Our approach is complementary to the conventional multiresolution approaches that rely on speeding up the rendering by reducing the number of graphics primitives for display.

In this paper we have mostly dealt with reducing the precision of all the three coordinates x, y, and z of the displayed primitives. However, a case can be made for not reducing the precision for the eye-space z in applications that have a high depth complexity since it can interfere with the proper execution of the visible-surface determination stage of the graphics pipeline (e.g., see [25]). This suggests potential for further work in relating precision in x, y, z with accurate scan-conversion, extending the precision work in [25] and in this paper.

Here we have focused on using variable precision for transformations and lighting. In future, one can carry out a similar analysis to extend it to texture coordinate computation as well. Also interesting will be applications of this method to compute the delta-differences on-the-fly for hardware-supported graphics primitives such as triangle strips that currently only exploit connectivity coherence, but using ideas in this paper could also benefit from spatial coherence. We plan to explore these ideas in the future.

10 Acknowledgements

We would like to acknowledge the reviewers for their very detailed and constructive comments which have led to a much better presentation of our results. This work has been supported in part by the NSF grants: IIS-00-81847, ACR-98-12572, and DMI-98-00690.

References

[1] C. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages 307 – 316, Los Alamitos, California, 1999. IEEE, Computer Society Press.

[2] C. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. *Computational Geometry: Theory and Applications*, vol 14:167–186, 2000.

[3] R. Banerjee and J. Rossignac. Topologically exact evaluation of polyhedra defined in CSG with loose primitives. *Computer Graphics Forum*, 15, No. 4:205–217, 1996.

[4] M. Chow. Optimized geomerty compression for real-time rendering. In *IEEE Visualization '97 Proceedings*, pages 403 – 410. ACM Press, October 1997.

[5] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17, No. 2:167–174, June 1998.

[6] D. Cohen-Or, D. Levin, and O. Remez. Progressive compression of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages 67–72, Los Alamitos, California, 1999. IEEE, Computer Society Press.

[7] M. F. Deering. Geometry compression. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 13–20. ACM SIGGRAPH, Addison Wesley. Los Angeles, California, August 1995.

[8] S. Fortune. Vertex rounding a three-dimensional polyhedral subdivision. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 116–125. ACM Press, June 1998.

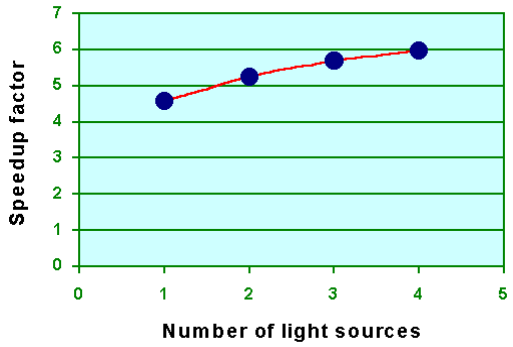


Figure 8: Speedup Factor as Function of Number of Light Sources (Venus Model)

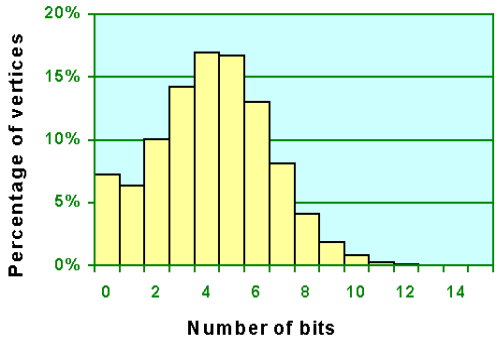
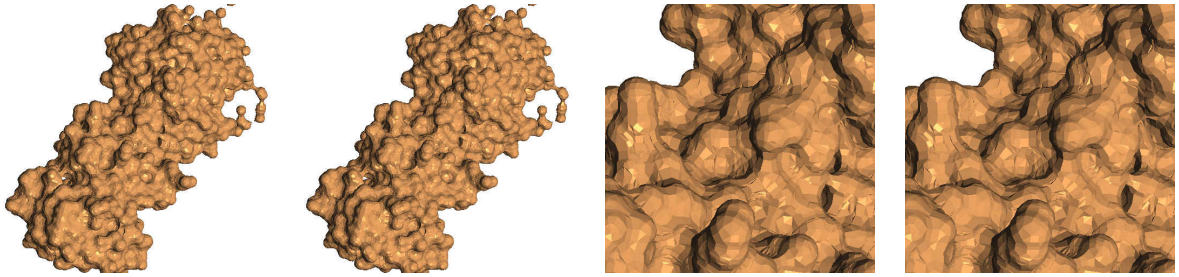
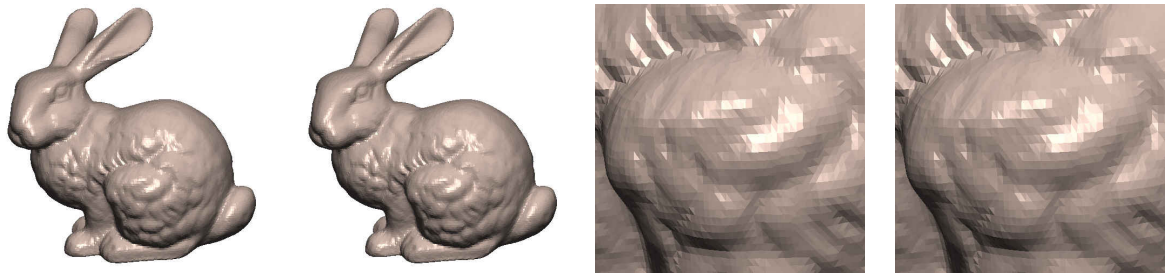


Figure 9: Histogram of Vertices Transformed in Different Number of Bits using Variable Precision (AMR Model)



(a) Floating Point (32 bits/Vertex Coordinate) (b) Variable Precision (7.9 bits/Vertex Coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 10: Dihydrofolate Reductase Molecular Surface (145K triangles) Rendered in Variable Precision



(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (7.9 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 11: Stanford Bunny (69K triangles) Rendered in Variable Precision

[9] S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[10] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 171–180. Addison Wesley, August 1996.

[11] P. Heckbert. Color image quantization for frame buffer display. In *Computer Graphics (SIGGRAPH 82 Procs)*, volume 16(3), pages 297–307, July 1982.

[12] J. Hirshon. A guide to CPU 3D instruction sets, August 1999. http://www.3d-design.com /newsletter /1999 /0899/horizon0899.html.

[13] C. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, California, 1989.

[14] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH 97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 189 – 197. ACM Press, August 1997.

[15] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *15th ACM Symp. on Computational Geometry*, 1999.

[16] D. King and J. Rossignac. Optimal bit allocation in compressed 3d models. *Journal of Computational Geometry, Theory and Applications*, 14:91–118, November 1999.

[17] J. Li and C. C. Kuo. Progressive coding of 3D graphics models. *Proceedings of the IEEE*, 86(6):1052–1063, June 1998.

[18] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings (Los Angeles, CA)*, Annual Conference Series, pages 198 – 208. ACM Press, August 1997.

[19] V. Milenkovic and L. Nackman. Finding compact coordinate representations for polygons and polyhedra. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 244–252. ACM Press, June 1990.

[20] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January 2000.

[21] R. Pajarola and J. Rossignac. Squeeze: Fast and progressive decompression of triangle meshes. In *Proceedings Computer Graphics International CGI 2000*, pages 173–182, Los Alamitos, California, 2000. IEEE, Computer Society Press.

[22] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[23] F. Pellacini, J. A. Ferwerda, and D. P. Greenberg. Toward a psychophysically-based light reflection model for image synthesis. In *Siggraph 2000 Conference Proceedings*, Annual Conference Series, pages 55–64. ACM Press, 2000.

[24] S. M. Pizer and V. L. Wallace. *To Compute Numerically: Concepts and Strategies*. Little, Brown Computer. Systems Series, 1983.

[25] J. Rossignac. Accurate scan-conversion of triangulated surfaces. In *Advances in Computer Graphics Hardware VI*, Proc. of the 6th Eurographcis Workshop on Computer Graphics Hardware, pages 13–20. Springer Verlag. September 1991.

[26] J. Rossignac. Edgebreaker: Connnectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1), January 1999.

[27] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June 1993.

[28] K. Sugihara. On finite-precision representations of geometric objects. *Journal of Computer and System Sciences*, 39:236–247, 1989.

[29] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 123–132. ACM SIGGRAPH, 1998.

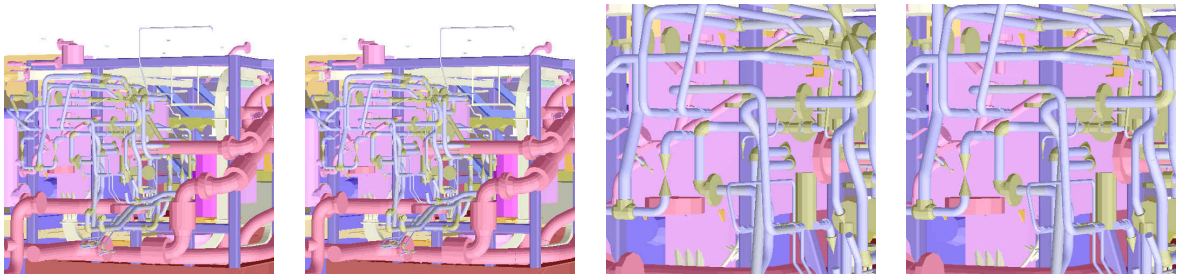
[30] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[31] B. Taylor and C. E. Kuyatt. Guidelines for evaluating and expressing the uncertainty of NIST measurement results. Technical Report Technical Note 1297, National Institute of Standards and Technology, Gaithersburg, MD, January 1993.

[32] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3, No. 2:171 – 183, June 1997.

[33] Z. Xiang. Color image quantization by minimizing the maximum intercluster distance. *ACM Transactions on Graphics*, 16(3):260–276, July 1997.

[34] H. Zhang and K. E. Hoff. Fast backface culling using normal masks (color plate S. 189). In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 103–106, New York, April27–30 1997. ACM Press.

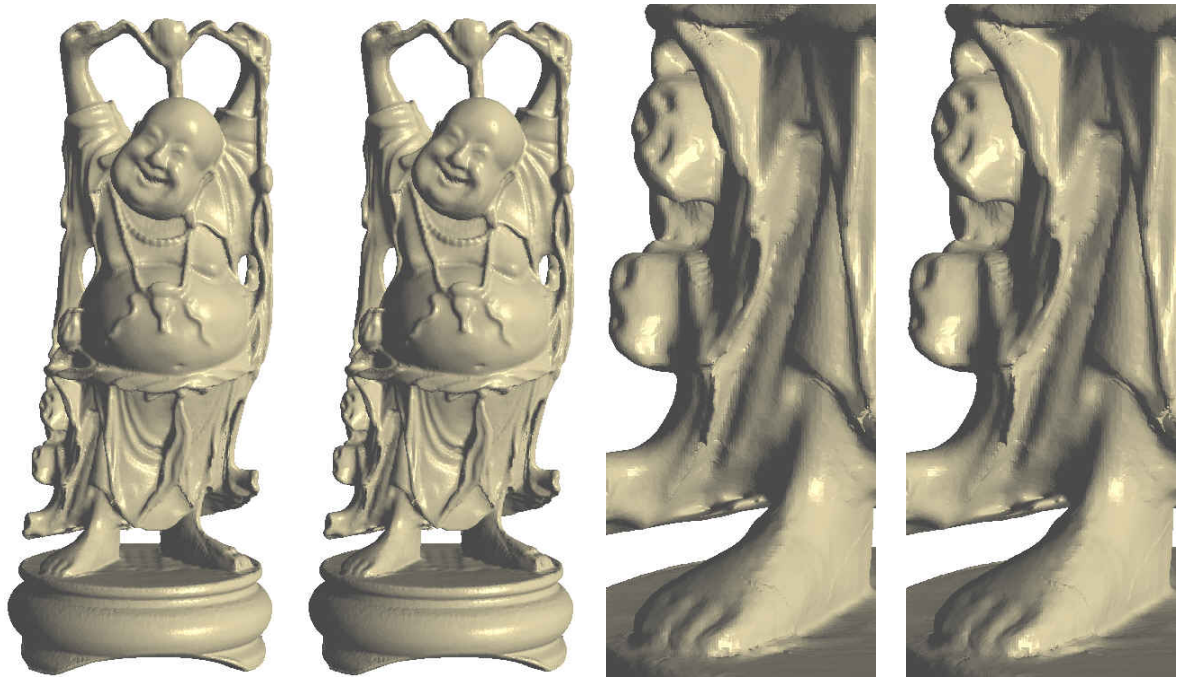


(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (4.2 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 12: Auxiliary Machine Room (376K triangles) Rendered in Variable Precision

(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (7.1 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 13: Cyberware Venus (268K triangles) Rendered in Variable Precision



(a) Floating Point (32 bits/vertex coordinate) (b) Variable Precision (5.6 bits/vertex coordinate) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 14: Buddha Model(1087K triangles) Rendered in Variable Precision