Geometry Compression

Michael Deering

Sun Microsystems[†]

ABSTRACT

This paper introduces the concept of Geometry Compression, allowing 3D triangle data to be represented with a factor of 6 to 10 times fewer bits than conventional techniques, with only slight losses in object quality. The technique is amenable to rapid decompression in both software and hardware implementations; if 3D rendering hardware contains a geometry decompression unit, application geometry can be stored in memory in compressed format. Geometry is first represented as a generalized triangle mesh, a data structure that allows each instance of a vertex in a linear stream to specify an average of two triangles. Then a variable length compression is applied to individual positions, colors, and normals. Delta compression followed by a modified Huffman compression is used for positions and colors; a novel table-based approach is used for normals. The table allows any useful normal to be represented by an 18-bit index, many normals can be represented with index deltas of 8 bits or less. Geometry compression is a general space-time tradeoff, and offers advantages at every level of the memory/interconnect hierarchy: less storage space is needed on disk, less transmission time is needed on networks.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation *Display algorithms*; I.3.7 [Computer Graphics]: Three Dimensional Graphics and Realism.

Additional Keywords and Phrases: 3D graphics hardware, compression, geometry compression.

1 INTRODUCTION

Modern 3D computer graphics makes extensive use of geometry to describe 3D objects. Many graphics techniques are available for such use. Complex smooth surfaces can be succinctly represented by high level abstractions such as trimmed NURBS. Detailed surface geometry can many times be rendered by use of texture maps. But as realism is added, more and more raw geometry is required, usually in the form of triangles. Position, color, and normal components of these triangles are typically represented as floating point numbers; an isolated triangle can take on the order of 100 bytes or more of storage to describe. To maximize detail while minimizing the number of triangles, triangle re-tessellation techniques can be employed. The techniques described in the current paper are complementary: for a fixed number of triangles, minimize the total bit-size of the representation, subject to quality (and implementation) trade-offs.

While many techniques exist for (lossy and lossless) compression of 2D pixel images, and at least one exists for 2D geometry [2], no corresponding techniques have previously been available for compression of 3D triangles. This paper describes a viable algorithm for Geometry Compression, which furthermore is suitable for implementation in real-time hardware. The availability of a decompression unit within rendering hardware means that geometry can be stored and transmitted entirely in compressed format. This addresses one of the main bottlenecks in current graphics accelerators: input bandwidth. It also greatly increases the amount of geometry that can be cached in main memory. In distributed networked applications, compression can help make shared VR display environments feasible, by greatly reducing transmission time. Even low-end video games are going true 3D with a vengeance, but without compression even CD-ROMs are limited to a few tens of millions of triangles total storage.

The technique described here can achieve (lossy) compression ratios of between 6 and 10 to 1, depending on the original representation format and the final quality level desired. The compression proceeds in four stages. The first is the conversion of triangle data into a generalized triangle mesh form. The second is the quantization of individual positions, colors, and normals. Quantization of normals includes a novel translation to non-rectilinear representation. In the third stage the quantized values are delta encoded between neighbors. The final stage performs a Huffman tag-based variable-length encoding of these deltas. Decompression is the reverse of this process; the decompressed stream of triangle data is then passed to a traditional rendering pipeline, where it is processed in full floating point accuracy.

2 REPRESENTATION OF GEOMETRY

Today, most major MCAD and many animation modeling packages allow the use of CSG (constructive solid geometry) and free-form NURBS in the construction and representation of geometry. The resulting trimmed polynomial surfaces are a high-level representation of regions of smooth surfaces. However for hardware rendering, these surfaces are typically pre-tessellated in software into triangles prior to transmission to the rendering hardware, *even on hardware that supports some form of hardware NURBS rendering*. Furthermore, much of the advantage of the NURBS representation of geometry is for tasks other than real-time rendering. These non-ren-

[†]2550 Garcia Avenue, UMPK14-202
Mountain View, CA 94043-1100
michael.deering@Eng.Sun.COM (415) 786-6325

dering tasks include representation for machining, physical analysis (for example, simulation of turbulence flow), and interchange. Also, accurately representing the trimming curves for NURBS is quite data intensive; as a compression technique, trimmed NURBS can be not much more compact than pre-tessellated triangles, at least at typical rendering tessellation densities. Finally, not all objects are compactly represented by NURBS; outside the mechanical engineering world of automobile hoods and jet turbine blades, the entertainment world of tiger's teeth and tennis shoes do not have large, smooth areas where NURB representations would have any advantage. Thus while NURBS will continue to be appropriate in many cases in the modeling world, compressed triangles will be far more compact for many classes of application objects.

For many years photorealistic batch rendering has made extensive use of texture map techniques (color texture maps, normal bump maps, displacement maps) to compactly represent fine geometric detail. With texture mapping support starting to appear in rendering hardware, real-time renders can also apply these techniques. Texture mapping works quite well for large objects in the far background: clouds in the sky, buildings in the distance. At closer distances, textures work best for 3D objects that are mostly flat: billboards, paintings, carpets, marble walls, etc. But for nearby objects that are not flat, there is a noticeable loss of quality. One technique is the "signboard", where the textured polygon always swivels to face the observer. But this technique falls short: when viewed in stereo, especially head-tracked virtual reality stereo, nearby textures are plainly perceived as flat. Here even a lower detail but fully three dimensional polygonal representation of a nearby object is much more realistic. Thus geometry compression and texture mapping are complementary techniques; each is more appropriate for a different portion of a scene. What is important to note is that geometry compression achieves the *same or better* representation density as texture mapping. In the limit they are the same thing; in the Reyes rendering architecture [1] deformation mapped texels are converted into micro-polygons before being rendered.

Since the very early days of 3D raster computer graphics, polyhedral representation of geometry has been supported. Specified typically as a list of vertices, edges, and faces, arbitrary geometry can be expressed. These representations, such as winged-edge data structures (cf. [6]), were as much designed to support editing of the geometry as display. Nowadays vestiges of these representations live on as interchange formats (for example, Wavefront OBJ). While theoretically compact, some of the compaction is given up for readability by use of ASCII representation of the data in interchange files. Also, few of these formats are set up to be directly passed to rendering hardware as drawing instructions. Another historical vestige is the support of n-sided polygons in such formats. While early rendering hardware could accept such general primitives, nearly all of today's (very much faster) hardware mandates that all polygon geometry be reduced to triangles before being submitted to hardware. Polygons with more than three sides cannot in general be guaranteed to be either planar or convex. If quadrilaterals are accepted as rendering primitives, the fine print somewhere indicates that they will be (arbitrarily) split into a pair of triangles before rendering. In keeping with this modern reality, we restrict geometry to be compressed to triangles.

Modern graphics languages specify binary formats for the representation of collections of 3D triangles, usually as arrays of vertex data structures. PHIGS PLUS, PEX, XGL, and proposed extensions to OpenGL are of this form. These formats define the storage space taken by executable geometry today.

Triangles can be isolated or chained in "zig-zag" or "star" strips. IrisGL, XGL, and PEX 5.2 define a form of generalized triangle strip that can switch from a zig-zag to star-like vertex chaining on a vertex by vertex basis (at the expense of an extra header word per vertex in XGL and PEX). In addition, a restart code allows multiple disconnected strips of triangles to be specified within one array of vertices. In these languages, all vertex components (positions, colors, normals) may be specified by 32-bit single precession IEEE floating point numbers, or 64-bit double precision numbers. XGL, IrisGL, and OpenGL also have some 32-bit integer support. IrisGL and OpenGL support input of vertex position components as 16-bit integers; normals and colors can be any of these as well as 8-bit components.

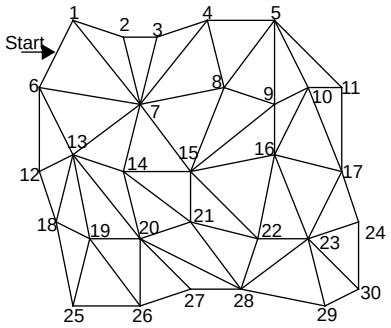
As will be seen, positions, colors, and normals can be quantized to significantly fewer than 32 bits (single precision IEEE floating point), with little loss in visual quality. Indeed, such bit-shaving can be utilized in commercial 3D graphics hardware, so long as supported by appropriate numerical analysis (cf. [3][4]).

3 GENERALIZED TRIANGLE MESH

The first stage of geometry compression is to convert triangle data into an efficient linear strip form: the *generalized triangle mesh.* This is a near-optimal representation of triangle data, given fixed storage.

The existing concept of a generalized triangle strip structure allows for compact representation of geometry while maintaining a linear data structure. That is, the geometry can be extracted by a single monotonic scan over the vertex array data structure. This is very important for pipelined hardware implementations, a data format that requires random access back to main memory during processing is very problematic.

However, by confining itself to linear strips, the generalized triangle strip format leaves a potential factor of two (in space) on the table. Consider the geometry in figure 1. While it can be represented by one triangle strip, many of the interior vertices appear twice in the strip. This is inherent in any approach wishing to avoid references to old data. Some systems have tried using a simple regular mesh buffer to support re-use of old vertices, but there is a problem with this in practice: in general, geometry does not come in a perfectly regular rectangular mesh structure.



Generalized Triangle Strip:
R6, O1, O7, O2, O3, M4, M8, O5, O9, O10, M11,
M17, M16, M9, O15, O8, O7, M14, O13, M6,
O12, M18, M19, M20, M14, O21, O15, O22, O16,
O23, O17, O24, M30, M29, M28, M22, O21, M20,
M27, O26, M19, O25, O18

Generalized Triangle Mesh:
R6p, O1, O7p, O2, O3, M4, M8p, O5, O9p, O10, M11,
M17p, M16p, M-3, O15p, O-5, O6, M14p, O13p, M-9,
O12, M18p, M19p, M20p, M-5, O21p, O-7, O22p, O-9,
O23, O-10, O-7, M30, M29, M28, M-1, O-2, M-3,
M27, O26, M-4, O25, O-5

Legend:
First letter: R = Restart, O = Replace Oldest, M = Replace Mi
Trailing "p" = push into mesh buffer
Number is vertex number, -number is mesh buffer reference
where -1 is most recent pushed vertex.

Figure 1. Generalized Triangle Mesh

The generalized technique employed by geometry compression addresses this problem. Old vertices are *explicitly* pushed into a queue, and then explicitly referenced in the future when the old vertex is desired again. This fine control supports irregular meshes of nearly any shape. Any viable technique must recognize that storage is finite; thus the maximum queue length is fixed at 16, requiring a 4-bit index. We refer to this queue as the *mesh buffer*. The combination of generalized triangle strips and mesh buffer references is referred to as a *generalized triangle mesh*.

The fixed mesh buffer size requires all tessellators/re-strippers for compressed geometry to break up any runs longer than 16 unique references. Since geometry compression is not meant to be programmed directly at the user level, but rather by sophisticated tessellators/re-formatters, this is not too onerous a restriction. Sixteen old vertices allows up to 94% of the redundant geometry to avoid being re-specified.

Figure 1 also contains an example of a general mesh buffer representation of the surface geometry.

The language of geometry compression supports the four vertex replacement codes of generalized triangle strips (replace oldest, replace middle, restart clockwise, and restart counterclockwise), and adds another bit in each vertex header to indicate if this vertex should be pushed into the mesh buffer or not. The mesh buffer reference command has a 4-bit field to indicate which old vertex should be re-referenced, along with the 2-bit vertex replacement code. Mesh buffer reference commands do *not* contain a mesh buffer push bit; old vertices can only be recycled once.

Geometry rarely is comprised purely of positional data; generally a normal and/or color are also specified per vertex. Therefore, mesh buffer entries contain storage for all associated per-vertex information (specifically including normal and color). For maximum space efficiency, when a vertex is specified in the data stream, (per vertex) normal and/or color information should be directly bundled with the position information. This bundling is controlled by two state bits: *bundle normals with vertices* (*bnv*), and *bundle colors with vertices* (*bcv*). When a vertex is pushed into the mesh buffer, these bits control if its bundled normal and/or color are pushed as well. During a mesh buffer reference command, this process is reversed; the two bits specify if a normal and/or color should be inherited from the mesh buffer storage, or inherited from the *current normal* or *current color*. There are explicit commands for setting these two current values. An important exception to this rule occurs when an explicit "set current normal" command is followed by a mesh buffer reference, with the bnv state bit active. In this case, the former overrides the mesh buffer normal. This allows compact representation of hard edges in surface geometry. The analogous semantics are also defined for colors, allowing compact representation of hard edges in textures.

Two additional state bits control the interpretation of normals and colors when the stream of vertices is turned into triangles. The *replicate normals over triangle* (*rnt*) bit indicates that the normal in the final vertex that completes a triangle should be replicated over the entire triangle. The *replicate colors over triangle* (*rct*) bit is defined analogously.

4 Compression of xyz Positions

The 8-bit exponent of 32-bit IEEE floating-point numbers allows positions literally to span the known universe: from a scale of 15 billion light years, down to the radius of sub-atomic particles. However for any given tessellated object, the exponent is really specified just once by the current modeling matrix; within a given modeling space, the object geometry is effectively described with only the 24-bit fixed-point mantissa. Visually, in many cases far fewer bits are needed; thus the language of geometry compression supports variable quantization of position data down to as little a one bit. The question then is what is the maximum number of bits that should be supported? Based on empirical visual tests we have done, as well as silicon implementation considerations, we decided to limit our implementation to support of at most 16 bits of precision per component of position.

We still assume that the position and scale of the local modeling spaces are specified by full 32-bit or 64-bit floating-point coordinates. If sufficient numerical care is taken, multiple such modeling spaces can be stitched together without cracks, forming seamless geometry coordinate systems with much greater than 16-bit positional precision.

Most geometry is local, so within the 16-bit (or less) modeling space (of each object), the delta difference between one vertex and the next in the generalized mesh buffer stream is very likely to b less than 16 bits in significance. Indeed one can histogram the bit length of neighboring position deltas in a batch of geometry, and based upon this histogram assign a variable length code to compactly represent the vertices. The typical coding used in many other similar situations is customized Huffman code; this is the case for geometry compression. The details of the coding of position deltas are postponed until section 7, where they can be discussed in the context of color and normal delta coding as well.

5 Compression of RGB Colors

We treat colors similar to positions, but with a smaller maximum accuracy. Thus rgbα color data is first quantized to 12-bit unsigned fraction components. These are absolute linear reflectivity values, with 1.0 representing 100% reflectivity. An additional parameter allows color data effectively to be quantized to any amount less than 12 bits, i.e. the colors can all be within a 5-5-5 rgb color space. (The α field is optional, controlled by the *color alpha present* (*cap*) state bit.) Note that this decision does *not* necessarily cause mock banding on the final rendered image; individual pixel colors are still interpolated between these quantized vertex colors, and also are subject to lighting.

After considerable debate, it was decided to use the same delta coding for color components as is used for positions. Compression of color data is where geometry compression and traditional image compression face the most similar problem. However, many of the more advanced techniques for image compression were rejected for geometry color compression because of the difference in focus.

Image compression (for example, JPEG [7]) makes several assumptions about the viewing of the decompressed data that *cannot* be made for geometry compression. In image compression, it is known a priori that the pixels appear in a perfectly rectangular array, and that when viewed, each pixel subtends a narrow range of visual angles. In geometry compression, one has almost no idea what the relationship between the viewer and the rasterized geometry will be.

In image compression, it is known that the spatial frequency on the viewer's eyes of the displayed pixels is likely higher than the human visual system's color acuity. This is why colors are usually converted to yuv space (cf. [6]), so that the uv color components can be represented at a lower spatial frequency than the y (intensity) component. Usually the digital bits representing the sub-sampled uv components are split up among two or more pixels. Geometry compression can't take advantage of this because the display scale of the geometry relative to the viewer's eye is not fixed. Also, given that compressed triangle vertices are connected to 4 - 8 or more other vertices in the generalized triangle mesh, there is no consistent way of sharing "half" the color information across vertices.

Similar arguments apply for the more sophisticated transforms used in traditional image compression, such as the discrete cosine transform. These transforms assume a regular (rectangular) sampling of pixel values, and require a large amount of random access during decompression.

Another traditional approach avoided was pseudo-color look-up tables. Any such look-up table would have to have a (fixed) maximum size, and yet still is a very expensive resource for real-time processing. While pseudo-color indices would result in a slightly

higher compression ratio for certain scenes, it was felt that the rgb model is more general and considerably less expensive.

Finally, the rgb values are represented as linear reflectance values. In theory, if all the effects of lighting are known ahead of time, a bit or two could have been shaved off the representation if the rgb components had been represented in a nonlinear, or perceptually linear (sometime referred to as gamma corrected) space. However, in general, the effects of lighting are not predictable, and considerable hardware resources would have had to be expended to convert from nonlinear to linear light on the fly.

6 COMPRESSION OF NORMALS

Probably the most innovative concept in geometry compression is the method of compressing surface normals. Traditionally 96-bit normals (three 32-bit IEEE floating-point numbers) are used in calculations to determine 8-bit color intensities. 96 bits of information theoretically could be used to represent 2^{96} different normals, spread evenly over the surface of a unit sphere. This is a normal every 2^{-46} radians in any direction. Such angles are so exact that spreading out angles evenly in every direction from earth you could point out any rock on Mars with sub-centimeter accuracy.

But for normalized normals, the exponent bits are effectively unused. Given the constraint $|N| = 1$, at least one of N_x, N_y, or N_z, *must* be in the range of 0.5 to 1.0. During rendering, this normal will be transformed by a composite modeling orientation matrix T: $N' = N \cdot T$.

Assuming the typical implementation in which lighting is performed in world coordinates, the view transform is not involved in the processing of normals. If the normals have been pre-normalized, then to avoid redundant re-normalization of the normals, the composite modeling transformation matrix T is typically pre-normalized to divide out any scale changes, and thus:

$$T_{0,0}^{\ 2} + T_{1,0}^{\ 2} + T_{2,0}^{\ 2} = 1, \text{ etc.}$$

During the normal transformation, floating-point arithmetic hardware effectively truncates all additive arguments to the accuracy of the largest component. The result is that for a normalized normal, being transformed by a scale preserving modeling orientation matrix, in all but a few special cases, the numerical accuracy of the transformed normal value is reduced to no more than 24-bit fixed-point accuracy.

Even 24-bit normal components are still much higher in angular accuracy than the (repaired) Hubble space telescope. Indeed, in some systems, 16-bit normal components are used. In [3] *9*-bit normal components were successfully used. After empirical tests, it was determined that an angular density of 0.01 radians between normals gave results that were not visually distinguishable from finer representations. This works out to about 100,000 normals distributed over the unit sphere. In rectilinear space, these normals still require high accuracy of representation; we chose to use 16-bit components including one sign and one guard bit.

This still requires 48 bits to represent a normal. But since we are only interested in 100,000 specific normals, in theory a single 17-bit index could denote any of these normals. The next section shows how it is possible to take advantage of this observation.

Normal as Indices

The most obvious hardware implementation to convert an index of a normal on the unit sphere back into a N_x N_y N_z value, is by table look-up. The problem is the size of the table. Fortunately, there are several symmetry tricks that can be applied to vastly reduce the size of the table (by a factor of 48). (In [5], effectively the same symmetries are applied to compress processed voxel data.)

First, the unit sphere is symmetrical in the eight quadrants by sign bits. In other words, if we let three of the normal representation bits be the three sign bits of the xyz components of the normal, then we only need to find a way to represent one eighth of the unit sphere.

Second, each octant of the unit sphere can be split up into six identical pieces, by folding about the planes $x = y$, $x = z$, and $y = z$. (See Figure 2.) The six possible sextants are encoded with another three bits. Now only 1/48 of the sphere remains to be represented.

This reduces the 100,000 entry look-up table in size by a factor of 48, requiring only about 2,000 entries, small enough to fit into an on-chip ROM look-up table. This table needs 11 address bits to index into it, so including our previous two 3-bit fields, the result is a grand total of 17 bits for all three normal components.

Representing a finite set of unit normals is equivalent to positioning points on the surface of the unit sphere. While no perfectly equal angular density distribution exists for large numbers of points, many near-optimal distributions exist. Thus in theory one of these with the same sort of 48-way symmetry described above could be used for the decompression look-up table. However, several additional constraints mandate a different choice of encoding:

1) We desire a scalable density distribution. This is one in which zeroing more and more of the low order address bits to the table still results in fairly even density of normals on the unit sphere. Otherwise a different look-up table for every encoding density would be required.

2) We desire a delta-encodable distribution. Statistically, adjacent vertices in geometry will have normals that are nearby on the surface of the unit sphere. Nearby locations on the 2D space of the unit-sphere surface are most succinctly encoded by a 2D offset. We desire a distribution where such a metric exists.

3) Finally, while the computational cost of the normal encoding process is not too important, in general, distributions with lower encoding costs are preferred.

For all these reasons, we decided to utilize a regular grid in the angular space within one sextant as our distribution. Thus rather than a monolithic 11-bit index, all normals within a sextant are *much* more conveniently represented as two 6-bit orthogonal angular addresses, revising our grand total to 18-bits. Just as for positions and colors, if more quantization of normals is acceptable, then these 6-bit indices can be reduced to fewer bits, and thus absolute normals can be represented using anywhere from 18 to as few as 6 bits. But as will be seen, we can delta encode this space, further reducing the number of bits required for high quality representation of normals.

Normal Encoding Parameterization

Points on a unit radius sphere are parameterized by two angles, θ and ϕ, using spherical coordinates. θ is the angle about the y axis;

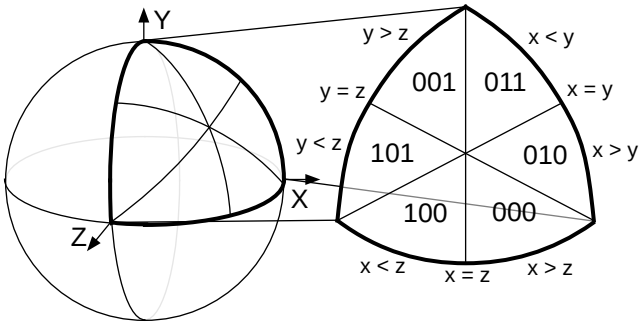


Figure 2. Encoding of the six sextants of each octant of a sphere.

ϕ is the longitudinal angle from the y=0 plane. The mapping between rectangular and spherical coordinates is:

$$x = \cos\theta \cdot \cos\phi \qquad y = \sin\phi \qquad z = \sin\theta \cdot \cos\phi \quad (1)$$

Points on the sphere are folded first by octant, and then by sort order of xyz into one of six sextants. All the table encoding takes place in the positive octant, in the region bounded by the half spaces:

$$x \geq z \qquad z \geq y \qquad y \geq 0$$

This triangular-shaped patch runs from 0 to $\pi/4$ radians in θ, and from 0 to as much as 0.615479709 radians in ϕ: ϕ_{max}.

Quantized angles are represented by two n-bit integers $\hat{\theta}_n$ and $\hat{\phi}_n$, where n is in the range of 0 to 6. For a given n, the relationship between these indices θ and ϕ is

$$\theta(\hat{\theta}_n) = \text{asin}\tan\left(\phi_{max} \cdot (n - \hat{\theta}_n)/2^n\right)$$
$$\phi(\hat{\phi}_n) = \phi_{max} \cdot \hat{\phi}_n/2^n \quad (2)$$

These two equations show how values of $\hat{\theta}_n$ and $\hat{\phi}_n$ can be converted to spherical coordinates θ and ϕ, which in turn can be converted to rectilinear normal coordinate components via equation 1.

To reverse the process, e.g. to encode a given normal N into $\hat{\theta}_n$ and $\hat{\phi}_n$, one cannot just invert equation 2. Instead, the N must be first folded into the canonical octant and sextant, resulting in N'. Then N' must be dotted with all quantized normals in the sextant. For a fixed n, the values of $\hat{\theta}_n$ and $\hat{\phi}_n$ that result in the largest (nearest unity) dot product define the proper encoding of N.

Now the complete bit format of absolute normals can be given. The uppermost three bits specify the octant, the next three bits the sextant, and finally two n-bit fields specify $\hat{\theta}_n$ and $\hat{\phi}_n$. The 3-bit sextant field takes on one of six values, the binary codes for which are shown in figure 2.

This discussion has ignored some details. In particular, the three normals at the corners of the canonical patch are multiply represented (6, 8, and 12 times). By employing the two unused values of the sextant field, these normals can be uniquely encoded as 26 special normals.

This representation of normals is amenable to delta encoding, at least within a sextant. (With some additional work, this can be extended to sextants that share a common edge.) The delta code between two normals is simply the difference in $\hat{\theta}_n$ and $\hat{\phi}_n$: $\Delta\hat{\theta}_n$ and $\Delta\hat{\phi}_n$.

7 COMPRESSION TAGS

There are many techniques known for minimally representing variable-length bit fields (cf. [7]). For geometry compression, we have chosen a variation of the conventional Huffman technique.

The Huffman compression algorithm takes in a set of symbols to be represented, along with frequency of occurrence statistics (histograms) of those symbols. From this, variable length, uniquely identifiable bit patterns are generated that allow these symbols to be represented with a near-minimum total number of bits, assuming that symbols do occur at the frequencies specified.

Many compression techniques, including JPEG [7], create unique symbols as tags to indicate the length of a variable-length data-field that follows. This data field is typically a specific-length delta value. Thus the final binary stream consists of (self-describing length) variable length tag symbols, each immediately followed by a data field whose length is associated with that unique tag symbol.

The binary format for geometry compression uses this technique to represent position, normal, and color data fields. For geometry compression, these <tag, data> fields are immediately preceded by (a more conventional computer instruction set) op-code field. These fields, plus potential additional operand bits, are referred to as *geometry instructions* (see figure 3).

Traditionally, each value to be compressed is assigned its own associated label, e.g. an xyz delta position would be represented by three tag-value pairs. However, the delta xyz values are *not* uncorrelated, and we can get both a denser and simpler representation by taking advantage of this fact. In general, the xyz deltas statistically point equally in all directions in space. This means that if the number of bits to represent the largest of these deltas is n, then statistically the other two delta values require an average of n-1.4 bits for their representation. Thus we made the decision to use a *single* field-length tag to indicate the bit length of Δx, Δy, *and* Δz. This also means that we cannot take advantage of another Huffman technique that saves somewhat less than one more bit per component, but our bit savings by not having to specify two additional tag fields (for Δy and Δz) outweigh this. A single tag field also means that a hardware decompression engine can decompress all three fields in parallel, if desired.

Similar arguments hold for deltas of rgbα values, and so here also a single field-length tag indicates the bit-length of the r, g, b, and α (if present) fields.

Both absolute and delta normals are also parameterized by a single value (n), which can be specified by a single tag.

We chose to limit the length of the Huffman tag field to the relatively small value of six bits. This was done to facilitate high-speed low-cost hardware implementations. A 64-entry tag look-up table allows decoding of tags in one clock cycle. Three such tables exist: one each for positions, normals, and colors. The tables contain the length of the tag field, the length of the data field(s), a data normalization coefficient, and an absolute/relative bit.

One additional complication was required to enable reasonable hardware implementations. As will be seen in the next section, all instruction are broken up into an eight-bit header, and a variable length body. Sufficient information is present in the header to determine the length of the body. But in order to give the hardware time to process the header information, the header of one instruction must be placed in the stream *before the body of the previous instruction*. Thus the sequence ... B0 H1B1 H2B2 H3 ... has to be encoded:

... H1 B0 H2 B1 H3 B2 ... .

8 GEOMETRY COMPRESSION INSTRUCTIONS

All of the pieces come together in the geometry compression instruction set, seen in figure 3.

The **Vertex** command specifies a Huffman compressed delta encoded position, as well as possibly a normal and/or color, depending on bundling bits (bnv and bcv). Two additional bits specify a vertex replacement code (rep); another bit controls mesh buffer pushing of this vertex (mbp).

The **Normal** command specifies a new current normal; the **Color** command a new current color. Both also use Huffman encoding of delta values.

The **Set State** instruction updates the five state bits: rnt, rct, bnv, bcv, and cap.

The **Mesh Buffer Reference** command allows any of the sixteen most recently pushed vertices (and associated normals and/or colors) to be referenced as the next vertex. A 2-bit vertex replacement code is also specified.

The **Set Table** command sets entries in one of the three Huffman decoding tables (Position, Normal, or Color) to the entry value specified.

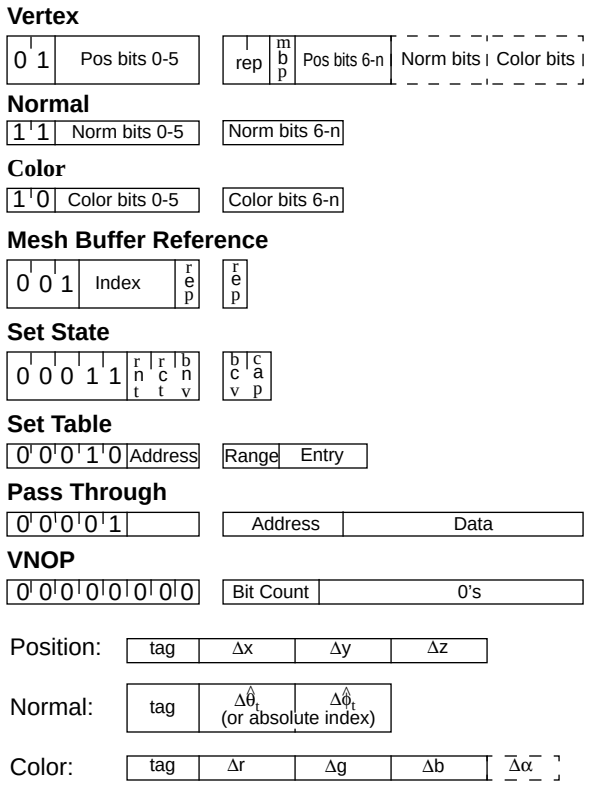


Figure 3. Geometry Compression Instruction Set

The **Pass Through** command allows additional graphics state not controlled directly by geometry compression to be updated in-line.

The **VNOP** (Variable length no-op) command allows fields within the bit stream to be aligned to 32-bit word boundaries, so that aligned fields can be patched at run-time.

9 RESULTS

The results are presented in figure 5 a-l and table 1. Figures 5 a-g are of the same base object (a triceratops), but with different quantization thresholds on positions and normals. Figure 5a is the original full floating-point representation: 96-bit positions, and 96-bit normals, which we denote by P96/N96. Figure 5b and 5c show the effects of purely positional quantization: P36/N96 and P24/N96, respectively. Figures 5d and 5e show only normal quantization: P96/N18 and P96/N12. Figures 5f and 5g show combined quantization: P48/N18 and P30/N36.

Figures 5 h-l show only the quantized results: for a galleon (P30/N12), a Dodge Viper (P36/N14), two views of a '57 Chevy (P33/N13), and an insect (P39/N15).

Without zooming into the object, positional quantization much above 24-bits has virtually no significant visible effect. As the normal quantization is reduced, the positions of specular highlights on the surfaces are offset slightly, but it is not visually apparent that these changes are reductions in quality, at least above 12 bits per normal. Note that the quantization parameters were photographed with the objects: otherwise even the author was not able to distinguish between the original and most compressed versions.

Compression (and other) statistics on these objects are summarized in table 1. The final column shows the compression ratios achieved over existing executable geometry formats. While the total byte count of the compressed geometry is an unambiguous number (and shown in the penultimate column), to state a compression ratio, some assumptions must be made about the object's uncompressed executable representation. We assumed optimized generalized triangle strips, with both positions and normals represented by floating-point values. This is how the "original size" column was calculated. To see the effect of pure 16-bit fixed point simple strip representation, we also show the byte count for this mode of OpenGL (the average strip length went way down, in the range of 2-3). Because few if any commercial products take advantage of generalized triangle strips, the potential memory space savings are considerably understated by the numbers in the table.

The earlier columns in the table break down the bit usage by component: just position tag/data, just normal tag/data, and everything else (overhead). The "quant" columns show the quantization thresholds. All results in table 1 are (measured) actual compression, with one exception. Because our software compressor does not yet implement a full meshifying algorithm, we present estimated mesh buffer results in parentheses (always next to actual results). This estimate assumes a 42% hit ratio in the mesh buffer.

While certainly there is statistical variation between objects (with respect to compression ratios), we have noted some general trends. When compressing using the highest quality setting of the quantization knobs (P48/N18), the compression ratios are typically about 6. When most objects start showing visible quantization artifacts, the ratios are nearly 10.

10 GEOMETRY COMPRESSION SOFTWARE

So far the focus has been on the justification and description of the geometry compression format. This section addresses some of the issues that arise when actually performing the compression; the next section addresses issues related to hardware and software implementation of decompression.

An important measure for any form of compression is the ratio of the time required for compression relative to decompression. Several otherwise promising techniques for image compression have failed in the marketplace because they require several thousand times more time to compress than to decompress. It is acceptable for off-line image compression to take 60X more time than decompression, but not too much more; for real-time video conferencing the ratio should be 1.

Geometry compression *does not* have this real-time requirement. Even if geometry is being constructed on the fly, most techniques for creating geometry (such as CSG) take orders of magnitude more time than displaying geometry. Also, unlike the continuous images found in movies, in most applications of geometry compression a compressed 3D object will be displayed for many sequential frames before being discarded. If the 3D object needs to be animated, this is typically done with modeling matrices. Indeed for a CD-based game, it is quite likely that an object will be decompressed billions of times by customers, while compressed only once by the authoring company.

Like some other compression systems, geometry compression algorithms can have a compression-time vs. compression-ratio knob. Thus for a given target level of quality, the more time allowed for compression, the better the compression ratio that can be achieved by a geometry compression system. There is a corresponding knob for quality of the resulting compressed 3D object. The lower the quality knob, the better the compression ratio achieved.

We have found an esthetic judgment involved in geometry compression, based upon our experiences with the system so far. Some 3D objects start to look bad when the target quantization of normals and/or positions is reduced even a little, others are visually unchanged even with a large amount of quantization. Sometimes the compression does cause visible artifacts, but may only make the object look different, not necessarily lower in quality. Indeed in one case an elephant started looking better (more wrinkled skin) the more we quantized his normals! The point is that there is also a subjective compo-

nent to geometry compression. In any highly compressed case, the original artist or modeling person that created the 3D object should also pass (interactive) judgment on the visual result of the compression. He or She alone can really say if the compressed object has captured the spirit of the original intent in creating the model.

But once a model has been created *and* compressed, it can be put into a library, to be used as 3D clip-art at the system level.

Below is an outline of the geometry compression algorithm:

1. Input explicit bag of triangles to be compressed, along with quantization thresholds for positions, normals, and colors.
2. Topologically analyze connectivity, mark hard edges in normals and/or color.
3. Create vertex traversal order & mesh buffer references.
4. Histogram position, normal, and color deltas.
5. Assign variable length Huffman tag codes for deltas, based on histograms, separately for positions, normals, colors.
6. Generate binary output stream by first outputting Huffman table initializations, then traversing the vertices in order, outputting appropriate tag and delta for all values.

Implementation status: a compressor of Wavefront OBJ format has been implemented. It supports compression of positions and normals, and creates full generalized triangle strips, but does not yet implement a full meshifying algorithm. The geometry compression format supports many more sophisticated compression opportunities than our existing compressor utilizes. We hope in the future to explore variable precision geometry, and fine structured updates of the compression tables. Eventually modelers should generate compressed geometry directly; our current compressor spends a lot of time figuring out geometric details that the tessellator already knew. The current (un-optimized) software can compress ~3K tris/sec.

11 Geometry Decompression Hardware

While many of the techniques employed by geometry compression are universal, some of the details were specifically designed to allow-cost, high-speed hardware implementations. A geometry compression format designed purely for software decompression would, of course, be a little different.

The features that make the geometry compression instruction set amenable to hardware implementation include: one pass sequential processing, limited local storage requirements, tag look-up rather than usual Hamming bit-sequential processing, and most arithmetic is comprised of shifts, adds, and look-ups.

Below is an outline of the geometry decompression algorithm:

1. Fetch the rest of the next instruction, and the first 8 bits of the instruction after that.
2. Using the tag table, expand any compressed value fields to full precision.

3a. If values are relative, add to current value; otherwise replace.

3b. If mesh buffer reference, access old values.

3c. If other command, do housekeeping.

4. If normal, pass index through ROM table to obtain full N_x N_y N_z values.
5. Output values in generalized triangle strip form to next stage.

Implementation status: a software decompressor has been implemented, and successfully decompresses compressed geometry, at a rate of ~10K triangles/second. Hardware designs are in progress, a simplified block diagram can be seen in figure 4.

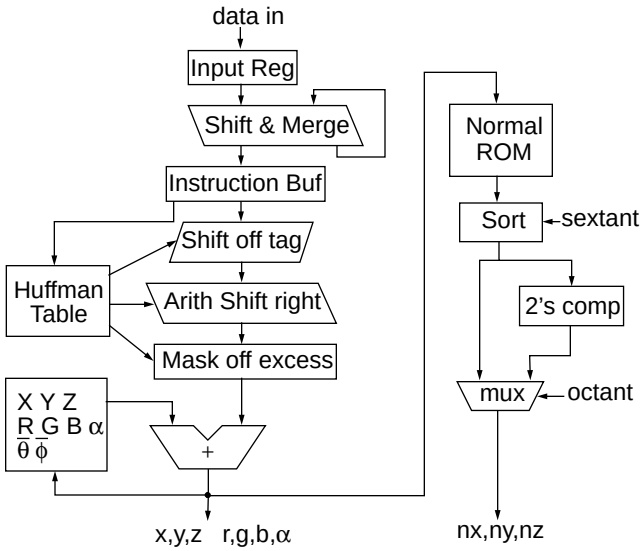


Figure 4. Decompression hardware block diagram (simplified).

12 Conclusions

A new technique for (lossy) compression of 3D geometric data has been presented. Compression ratios of 6 to 10 to one are achievable with little loss in displayed object quality. The technique has been designed for the constraints of low cost inclusion into real-time 3D rendering hardware, but is also of use in pure software implementations.

Acknowledgements

The author would like to thank Aaron Wynn for his work on the hardware and the meshifier, Michael Cox for help with the writing, Scott Nelson for comments on the paper and help with figures 1 & 2, and Viewpoint DataLabs for the 3D objects used in figure 5.

References

1. Cook, Robert, L. Carpenter, and E. Catmull. The Reyes Image Rendering Architecture. Proceedings of SIGGRAPH '87 (Anaheim, CA, July 27-31, 1987). In *Computer Graphics* 21, 4 (july 1987), 95-102.

2. Danskin, John. *Compressing the X Graphics Protocol*, Ph.D. Thesis, Princeton University, 1994.

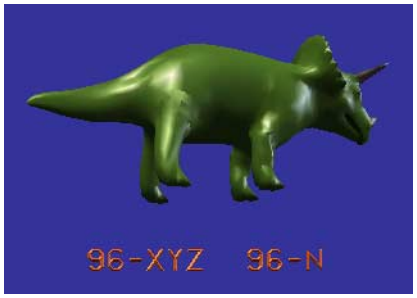
3. Deering, Michael, S. Winner, B. Schediwy, C. Duffy and N. Hunt. The Triangle Processor and Normal Vector Shader: A VLSI system for High Performance Graphics. Proceedings of SIGGRAPH '88 (Atlanta, GA, Aug 1-5, 1988). In *Computer Graphics* 22, 4 (July 1988), 21-30.

4. Deering, Michael, and S. Nelson. Leo: A System for Cost Effective Shaded 3D Graphics. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics* (August 1993), 101-108.

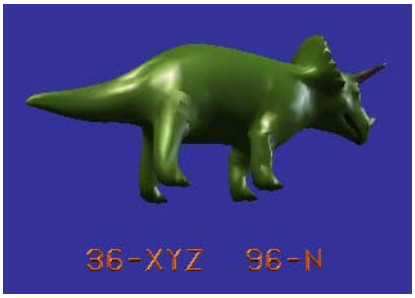
5. Durkin, James, and J. Hughes. *Nonpolygonal Isosurface Rendering for Large Volume Datasets*. Proceedings of Visualization '94, IEEE, 293-300.

6. Foley, James, A. van Dam, S. Feiner and J Hughes. *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley, 1990.

7. Pennebaker, William, and J. Mitchell. *JPEG Still Image Compression Standard*, Van Nostrand Reinhold, 1993.

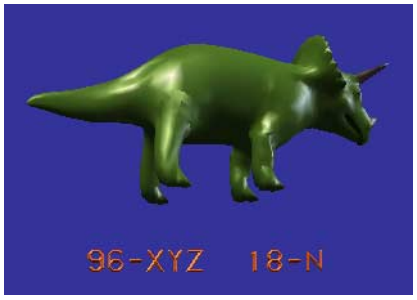


5a

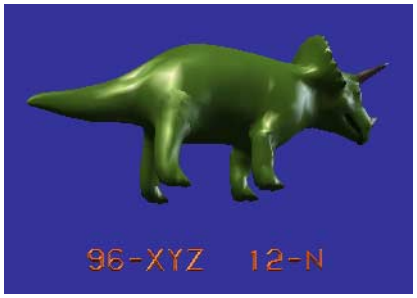


5b

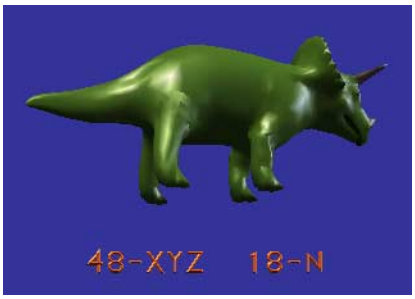
5c



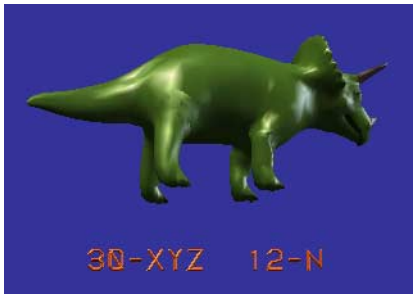
5d



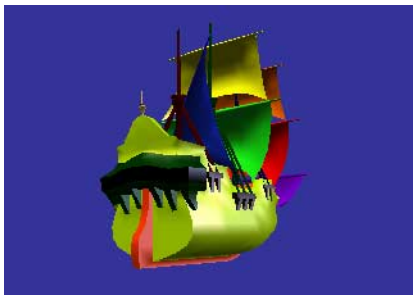
5e



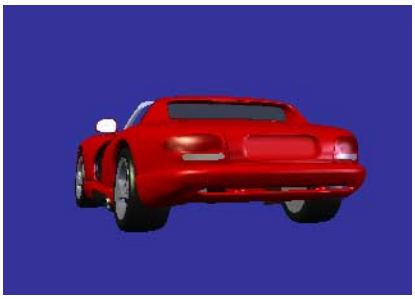
5f



5g



5h



5i

5j

5k

5l

Table 1:

object name	#Δ's	Δstrip length	overhead/ vertex	xyz quant	bits/ xyz	norm quant	bits/ norm	bits/tri	original size (bytes)	OpenGL 16-bit comp	compressed size (bytes)	compression ratio
triceratops	6,039	15.9	8.2	48	35.8	18	16.9	61.3 (41.2)	179,704	151,032	46,237 (31,038)	3.9X (5.8X)
triceratops	6,039	15.9	8.2	30	17.8	12	11.0	36.0 (26.5)	179,704	151,032	27,141 (19,959	6.7X (9.1X)
galleon	5,118	12.2	8.2	30	22.0	12	11.0	41.3 (29.7)	155,064	105,936	26,358 (18,954)	5.9X (8.2X)
viper	58,203	23.8	8.2	36	20.1	14	10.9	37.5 (27.2)	1,698,116	1,248,492	272,130 (197,525)	6.3X (8.6X)
57chevy	31,762	12.9	8.2	33	17.3	13	10.9	35.8 (26.4)	958,160	565,152	141,830 (104,691)	6.8X (9.2X)
insect	229,313	3.3	8.7	39	26.3	15	12.8	58.7 (40.9)	8,383,788	5,463,444	1,680,421 (1,170,237)	5.0X (7.2X)