Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

Hardware-Compatible Vertex Compression Using Quantization and Simplification

Budirijanto Purnomo, Jonathan Bilodeau, Jonathan D. Cohen, and Subodh Kumar

Johns Hopkins University

(a) Original (896 KB) (b) Our result at 64 bpv (224 KB) (c) DirectX-supported quantization at 96 bpv (336 KB) (d) OpenGL-supported quantization at 96 bpv (336 KB)

Figure 1: *Comparing our quantization result for NVIDIA RocketCar model to the standard API-supported quantizations. We can achieve a higher compression rate while maintaining higher image quality than other standard methods.*

Abstract

We present a vertex compression technique suitable for efficient decompression on graphics hardware. Given a user-specified number of bits per vertex, we automatically allocate bits to vertex attributes for quantization to maximize quality, guided by an image-space error metric. This allocation accounts for the constraints of graphics hardware by packing the quantized attributes into bins associated with the hardware's vectorized vertex data elements. We show that this general approach is also applicable if the user specifies a total desired model size. We present an algorithm that integrally combines vertex decimation and attribute quantization to produce the best quality model for a user-specified data size. Such models have an appropriate balance between the number of vertices and the number of bits per vertex.

Vertex data is transmitted to and optionally stored in video memory in the compressed form. The vertices are decompressed on-the-fly using a vertex program at rendering time. Our algorithms not only work well within the constraints of current graphics hardware but also generalize to a setting where these constraints are relaxed. They apply to models with a wide variety of vertex attributes, providing new tools for optimizing space and bandwidth constraints of interactive graphics applications.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques – Graphics data structures and data types I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Geometric algorithms, languages, and systems I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism E.4 [Coding and Information Theory]: Data Compaction and Compression

1. Introduction

Interactive graphics applications continue to push the limits of contemporary hardware. Even as that hardware becomes more powerful, developers continually strive to improve the quality of their applications to just within the limits of some appropriate level of interactivity. These quality improvements generally require more data and more bandwidth as well as more computation. The majority of this increased data is pushed through the graphics pipeline in the form of more vertices, more per-vertex attributes and more texture

images. Today's graphics cards have some native support for compression of texture data, but only the minimal support for compression of vertex data.

Although many sophisticated algorithms have been developed for geometry compression, most are incompatible with current graphics hardware. Similar to the adoption for texture compression of fixed-sized, block-based formats such as S3TC over more sophisticated formats such as JPEG, vertex data compression for graphics hardware requires an approach that is relatively simple and fast to decompress.

Our approach is based on a combination of attribute quantization and mesh simplification. In terms of quantization, we abstract the view of vertex attribute data presented by graphics drivers. In our presentation, each of the driver's vertex attribute data channels becomes a general-purpose bin, into which we may arrange bits of vertex attribute data in a more arbitrary way. This gives us more freedom to determine the relative bit sizes of the vertices' attributes.

With this increased flexibility in mind, we develop optimization algorithms for the following problems:

1. Given a desired number of total bits per vertex, determine an allocation of bits to vertex attributes to maximize rendering quality.
2. Given a desired number of total bits per vertex and a set of bins to arrange them in, determine an assignment of attributes to bins and an allocation of bits to attributes to maximize rendering quality.
3. Given a desired total storage size for a model, compute a simplification of the mesh vertices as well as an associated quantization to maximize rendering quality.

Our quality metric for the above optimizations is based on a comparison of rendered images over a sampled view space [LT00]. One advantage of this metric over attribute-space metrics is that it is capable of determining relative bit-allocations across a wide variety of logical attribute groups. This is especially important for today's 3D models, with their continually evolving, rich set of vertex attributes. Another advantage to this metric is that it can account for both the rendering algorithm applied to the 3D model and the 3D environment containing the particular model. Thus it is possible to create custom compressions for particular uses of a model.

2. Previous Work

Compression of both the topology and the vertex data has been an active area for research [TR98, AD01, Ise01, SKR01, KG02]. While much of the vertex compression research focuses on position data, normal and other attributes are also sometimes independently considered [Dee95]. Most compression algorithms start by quantizing the original set of vertices to a fixed point representation with B bits of precision, usually for some B less than 15. The resulting numbers are then compressed further, losslessly. This usually involves a prediction step based on the most recent few vertices decoded, followed by an entropy encoding of the residual. The prediction may be based on an offset [Dee95], a parallelogram rule [TG98, Ise02], a Fourier domain reconstruction [Tau95] or some other complex function. The results are sometimes as compact as to require only 4-5 bits to specify each vertex coordinate.

Spectral methods [KG00] consider the n vertices of a mesh as one element of an n-dimensional space for each coordinate (x, y and z). A projection of the n-dimensional space is then found and used to represent all the vertices in one block of bytes. Compression rates can be even higher, but it implies that all vertices must be decompressed together.

Compression of multi-resolution models may incur an additional degree of complexity [KSS00, PR00, AD01] but results are usually no more or less compact. Not only are most of these operations too complex for hardware, they usually require access to several other recently decoded vertices. Such random access is not conducive to hardware implementation.

Even though an early algorithm [Dee95, Cho97] was originally designed for hardware decompression, it still computes and encodes offsets between successive vertices in the vertex array. This requires that the vertex array be sequentially processed, which may not be necessarily the order implied by the element array. Furthermore, such sequential processing constraints are not suitable for parallel processing.

In a different approach, Hao and Varshney [HV01] reduce the geometry size and the rendering time by determining for each vertex the maximum precision needed for it. This computation is based on the number of pixels on the screen, the viewing parameters and the number of numerical operation performed on the vertex. While this approach reduces the number of needed bits without introducing noticeable errors, the generality of the method makes it difficult to implement on current hardware.

Calver describes basic principles for dealing with quantized vertex attributes in a vertex shader [Cal02, Cal04]. In addition to the necessary scaling and biasing, he demonstrates rotational transformation of coordinates to align a model's oriented bounding box with the coordinate axes. Such transformations can often be folded into the standard modeling transformation with little to no run-time cost.

Our vertex compression technique is suited for efficient decompression on current hardware. The run-time cost is higher than that of Calver, but the approach is more flexible, allowing more fine-tuned choice of quantization levels. An important constraint for hardware implementation that is violated by many other approaches is that each vertex must be decompressed independently. A vertex program operates on only a single vertex at a time, with no state retained in between. Like King and Rossignac [KR99], our approach also combines some degree of mesh simplification with quantization, although we consider not only geometry but all vertex attributes. We demonstrate the effectiveness of our compression and decompression using a vertex program based implementation. While the details of the implementation would indeed vary from generation to generation of graphics hard-

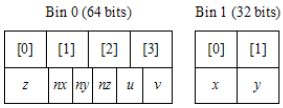


Figure 2: *Example quantization with allocation (x, y, z, nx, ny, nz, u, v) = (16, 16, 17, 9, 8, 9, 10, 11). Bins 0 and 1 have 4 and 2 16-bit components, respectively*

ware, the ideas presented here remain applicable and can be easily incorporated natively into the hardware architecture.

3. Overview

A primary goal of our compression scheme is to allow the compressed form of the vertex data to make it all the way to the graphics hardware, where it is decompressed at rendering time. Thus it is useful here to start by considering the target decompressor.

The decompression algorithm runs as a vertex program on the current generation of graphics hardware. This has several implications. First, each vertex must be processed independently; it cannot produce useful side effects to assist in decompression of other vertices. Second, the data channels we have available for transmitting the compressed vertex data are limited to the data channels exposed by current APIs. Furthermore, it is important to take advantage of the opportunity to store the compressed data in video memory on the far side of the bus.

Our decompression vertex program accepts as input for each vertex a sequence of bits which arrive packed into the 4-way vectorized floating point registers of the vertex unit. For a given chunk of vertices associated with a 3D model, constant parameters to the program specify a flexible blueprint, which specifies how the vertex parameters are packed into the data registers. This blueprint allows each *attribute* (e.g, x, y, z, nx, ny, nz, etc.) to have its own size in bits and also allows the individuals attributes of the various *attribute groups* (e.g. coordinate, normal, color, etc.) to be arbitrarily intermingled. The program unpacks the bits into the appropriate attribute variables, and scales and biases them from an integer coordinate system back into the application's desired floating point coordinate system. At this point, the vertex program can pick up whatever processing is required on the attributes (transformation, per-vertex lighting, etc.).

The implementation of unpacking in the vertex program is reasonably fast. Thus, if bandwidth is a bottleneck for some application, we can expect some speedup in rendering performance. This can happen, for example, if the original data is larger than the portion of video memory we wish to dedicate to geometry (as opposed to textures, etc.).

To generate the compressed data, we perform a bin-packed quantization procedure. Given a total number of bits per vertex, our algorithm allocates bits to individual attributes. These quantized attributes are allocated and arranged so as to fit in the vectorized data elements provided by the API. We can effectively think of each vectorized data element as a single bin, because our vertex program allows quantized attributes to cross the boundaries of the individual vector components, as illustrated in Figure 2.

In one variant of our decompressor, a single, general vertex program is used to unpack any bit arrangement for a given set of target attributes. This variant seems most amenable to acceleration by a custom hardware component, but restricts individual attributes from crossing the boundaries between different bins. Another variant eliminates this restriction efficiently, but employs custom vertex programs for each bit arrangement. Optionally, the user specifies a total bit size for the model, and our algorithm combines the quantization process with a mesh simplification process to produce an even higher quality model for a given size.

The output of our quantization method provides the vertex attributes in unsigned integer coordinates, as well as a scale and bias that may be used to convert these integers back into their original coordinate system. The quantized attributes are packed into the standard vectorized data elements accepted by the graphics driver and ultimately delivered to the vertex program with the blueprint for decompression into their target attribute groups.

4. Attribute Quantization

We automatically produce quantizations of 3D models using an optimization process. For this approach, we need to solve several problems. First, we need an error metric that can evaluate a proposed allocation of bits to attributes and produce a scalar error value. This metric should ideally be monotonic with respect to the number of bits assigned to each individual attribute. Second, we need an optimization process that, given a number of bits and a set of attributes, produces a good allocation of bits to attributes with respect to the error metric. Finally, we need to accommodate the constraints of the graphics hardware and of our vertex decompression program. This is accomplished by using a binned version of the preceding optimization process. We next describe these three sub-problems in more detail.

4.1. Image-Space Error Metric

Although a number of metrics have been used to quantify error for 3D models, few of them are immediately applicable to models with a variety of vertex attribute groups: coordinates, normals, colors, texture coordinates, blend weights, etc. Some of these metrics, such as multi-attribute error quadrics [GH98, Hop99], operate in the attribute space. They can quantify error appropriately within each attribute group, but the relative error between attribute groups is subject to some arbitrary weighting factors.

We have adopted the image-space metric first employed by Lindstrom and Turk [LT00] in the context of simplifying 3D meshes. This metric employs rendering to generate a number of images of the altered model, then compares these images to renderings of the original model. This approach has a number of benefits. It can handle arbitrary vertex attributes and employ any desired rendering algorithm and shaders, yet the metric itself is completely independent of

```
func ComputeBitAllocation(targetBPV)
  initialize attrib bit allocations
  while error decreases
    // Bit Reduce
    minError = MAXFLOAT
    for each attrib i
      attrib i bits -= 1
      error = ComputeError()
      if (error < minError)
        minError = error
        minAttrib = i
      attrib i bits += 1
    minAttrib bits -= 1

    // Bit Increase
    minError = MAXFLOAT
    for each attrib i
      attrib i bits += 1
      error = ComputeError()
      if (error < minError)
        minError = error
        minAttrib = i
      attrib i bits -= 1
    minAttrib bits += 1
return bit allocation
```

Figure 3: *Pseudocode for optimizing bit allocation across attributes*

these factors. One can use a variety of techniques to choose the parameters of sample renderings, and a variety of methods to compute the actual image-space error.

In our implementation, we use 20 image samples (placed on the 20 sides of an icosahedron surrounding the model) to measure the error of a particular quantization. Given the renderings of the quantized model and the associated renderings of the original model, we essentially subtract the two renderings for each image pair and take the root mean square of the difference image. We then average the rms errors for all the sample images to get the final error value. It has been shown that for comparing images of 3D models, rms error performs surprisingly well, although more sophisticated, perceptually-based metrics are possible [Lin00]. In our tests, we also find that the image resolution used makes little difference, so long as the triangles do not reach subpixel sizes and no texture minification occurs. For models intended for viewing across a range of viewing distances, it may be useful to measure the image error at different scales.

One aspect of this metric that is still left open to tweaking is the selection of a background color, which plays a significant role in determining the contribution of silhouette deviation to the error. By computing two renderings for the original objects using different background colors, we can actually distinguish the foreground (object) pixels from the background pixels and use this information in the final metric. For example, we can set some maximum error value for pixels which change between foreground and background in the two associated samples, or we can choose some medium error value. We always eliminate the contribution of pixels which are background in both the original and quantized renderings. This enables us to compare error values for different objects with different pixel coverages in a more meaningful way.

Interestingly, it is also possible to render the object in some larger 3D environment, with the background pixels coming from this environment. This effectively customizes the effect of silhouette deviation to the object's particular setting.

4.2. Metric Driven Quantization

Using a metric like the one above, we perform a greedy optimization process to allocate bits to vertex attributes (as shown in Figure 3).

We start with some initial guess of bit allocation that sums to the correct bit count. This guess may be a roughly even distribution of bits, or it may use some heuristics (e.g. assign twice as many bits to coordinate attributes as to normal attributes, assign each texture coordinate attribute the log of the texture resolution in bits, etc.). Then our optimization process repeatedly applies a bit-reduce and bit-increase operators to swap bits between attributes and reduce the error. When the error is no longer decreasing, the process terminates.

4.3. Adding Bin Constraints

The data paths from the CPU through the graphics API and to the graphics hardware impose some additional constraints on the allocation of bits to attributes. Normally, the API packages vertex attributes into vectorized data elements containing 1, 2, 3, or 4 components. Each such element contains the members of a single attribute group (e.g. x, y, and z for the coordinate group). Our vertex program relaxes this constraint, presenting each vectorized data element to us as a single bin that may contain any set of attributes we like. These attributes are free to cross the boundaries of the individual vector components, and attributes from different groups may be freely intermingled.

The constraints that remain are the following. First, an individual attribute may not span multiple bins. Although we eliminate this constraint in one variant of our decompressor, that variant may be less suitable for custom hardware acceleration. Second, the sizes of the bins are restricted according to the supported vector component types and the supported data transfer units.

For the current hardware, we restrict our bin sizes to be either 64 or 32 bits. In this case, we can always achieve the best results by using bins as large as possible, because it eases the constraint of attributes not spanning multiple bins (and can also increase vertex program performance).

In this binned context presented by the graphics hardware and our vertex program, we can now restate the quantization problem as follows. Given a 3D model and a user-specified number of bits per vertex (which should be a multiple of 32),

```
func ComputeBinnedBitAllocation(targetBPV)
   for each attribute-to-bin partitioning
      for each bin
         ComputeBitAllocation(binBPV)
return best partitioning and allocation
```

Figure 4: *Pseudocode for brute-force, binned bit allocation.*

	x	y	z	nx	ny	nz	u	v	Wasted bits	Error
Unrestricted (one bin)	16	15	17	9	8	9	11	11	0	0.0030
Binning (two bins: 64 and 32)	16	16	17	9	8	9	10	11	0	0.0031
DirectX standard	10	10	10	10	10	10	16	16	4	0.0210
OpenGL standard	8	8	8	8	8	8	16	16	16	0.0407

Figure 5: *Bit allocations for NVIDIA RocketCar model using 96 bits per vertex.*

compute the assignment of attributes to bins and the allocation of bits to each attribute such that the quality is maximized. The basic algorithm is shown in Figure 4.

Notice that if we are given a mapping of attributes to bins as well as the bin sizes, then each bin becomes an instance of the problem we have solved above in Section 4.2. We currently use a brute-force algorithm to examine all possible mappings of attributes to bins, then apply that optimization process to each bin. Given a attributes and b bins, the number of such mappings is b^a, where $b \leq a$. Due to the current constraints on bin sizes, there is no need to iterate over possible bin sizes as well; we assign sizes to the bins that are as large as possible, in decreasing order (e.g. 64, 64, 32 for a total bit size of 160 bits).

Our metric-optimized quantization process works well in the presence of the constraints imposed by current graphics hardware. Relaxing these constraints only improves the results further. Thus, the approach is quite general and should remain useful even on future hardware.

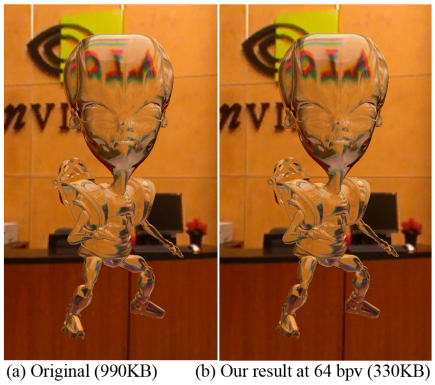
4.4. Results

We have tested our algorithm on a variety of models from NVIDIA demos and several models publicly available. As shown in Figure 1, we can use 64 bits per vertex (for geometry, normal and texture coordinate) for the NVIDIA RocketCar model compared to the standard quantization methods that are limited to using 96 bits per vertex (32 bits each for geometry, normal and texture coordinate). We can achieve another 33% compression compared to the standard quantization while producing similar/better image quality than the DirectX-supported quantization and noticeably better image quality than the OpenGL-supported quantization.

In Figure 5, our results produce higher image quality for a given bits per vertex compared to the naïve quantization supported by either DirectX or OpenGL approach. Our method

	Quantization method	x	y	z	nx	ny	nz	Error
Optimized for gouraud shading	Our method	14	15	15	7	7	6	0.007
	DirectX	10	10	10	10	10	10	0.026
	OpenGL	8	8	8	8	8	8	0.054
Optimized for environment map	Our method	13	14	13	8	8	8	0.009
	DirectX	10	10	10	10	10	10	0.022
	OpenGL	8	8	8	8	8	8	0.046

Figure 6: *Bit allocations for the NVIDIA HateAlien model using 64 bits per vertex with two different shaders and using the background from the original environment.*



(a) Original (990KB) (b) Our result at 64 bpv (330KB)

Figure 7: *A comparison of our quantization result at 64 bits per vertex versus the original.*

utilizes the entire 96 bits to attributes to maximize image quality. In Figure 6, we show that different shaders might generate different bit allocations. Here, we use the original environment as the background while computing the image metric. Figure 7 shows that at 64 bits per vertex (66% compression), our quantization result is indistinguishable from the rendering of the original model.

5. Combining Quantization and Simplification

The quantization algorithms described above perform a lossy compression of the 3D model while maximizing quality. It is possible, however, that many of the model's vertices do not contribute significantly to that quality. It is useful, then, to consider combining quantization with some degree of mesh simplification.

In this context, we redefine our compression problem statement slightly. Instead of the directly specifying the number of bits per vertex, the user specifies the total model output size (i.e. number of vertices times bits per vertex). Our goal, then, is to compute the simplification and quantization of the model that maximize the quality for the specified size.

5.1. Algorithm

Our algorithm for optimizing the simplification level along with the bit allocation is shown in Figure 8. Notice that we start with largest number of vertices and the smallest bits per vertex. At each iteration, we explore the space by increasing the bits per vertex and reducing the number of vertices

```
func QuantizeAndSimplifyModel(targetSize)
  currentBPV = targetSize/numVertices
  ComputeBinnedBitAllocation(currentBPV)
  while currentBPV is less than maxBPV
     currentBPV += hwCompatibleIncrement
     numVertices = targetSize/currentBPV
     simplify model to the numVertices
     ComputeBinnedBitAllocation(currentBPV)
     if (currentError < bestError)
       bestBPV = currentBPV
       bestError = currentError
return best BPV, error, allocation and model
```

Figure 8: *Pseudocode for combining simplification with bit allocation*

correspondingly. For current hardware, we generally must guarantee that the bits per vertex is a multiple of 32, but the algorithm is general enough to work for increments of 8 or even 1 bit per vertex.

In our implementation, we use a greedy, priority-queue-based, bottom-up simplification algorithm using a half-edge collapse operator [LRC*02]. Ideally, one would use the same image-space error metric to evaluate every individual half-edge collapse operation. However, to keep our implementation simple, we employ a geometric quadric error metric [GH97] during the simplification process, reserving evaluation of the image-space metric for the ensuing quantization.

The combined simplification and quantization process provides several benefits. First, it provides better quality renderings for a given storage size than quantization alone. (One might say that this is the first algorithm that actually uses mesh simplification to improve the quality of a model, by allowing a higher bit rate). Second, the reduction in vertex count eases the computational load on the vertex processing units. In principle, it should even be possible to design an optimization scheme that balances this reduction in vertices with the number of vertex program instructions added for the decompression process (the instruction count increases with the number of bins, and thus with the bit rate). We have not yet explored this last problem, but it seems quite interesting.

5.2. Results

In Figure 9, we show the two extremes of just using quantization and simplification alone versus our method of combining simplification and quantization for a given target size. The image error of our method is significantly lower than the two extremes. We illustrate this fact in Figure 10. We demonstrate results for several models in Figure 12.

As shown in Figure 11, we can find the optimal balance of quantization and simplification for a given storage size by choosing the bits per vertex that gives the least error in the graph. For the bunny model, 57 bits per vertex gives the least error. Fortunately, the error for the hardware-compatible 64-bit size is quite similar.

	Number of vertices	Bits per vertex	Error
Original (816 KB)	34834	192	0.0
Quantization only	34834	32	0.0558
Simplification only	5805	192	0.0336
Simplification and Quantization	17417	64	0.0155

Figure 9: *Three methods of compressing to 136KB storage size for the bunny model.*

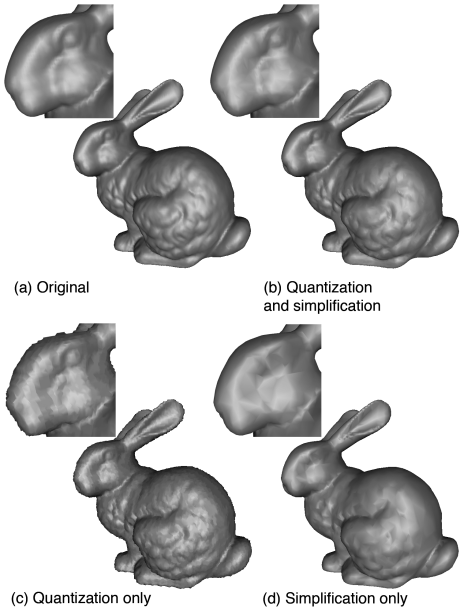


Figure 10: *Comparing the image quality of the combined quantization and simplification versus quantization alone and simplification alone. (b)-(d) has the same storage size of 136KB.*

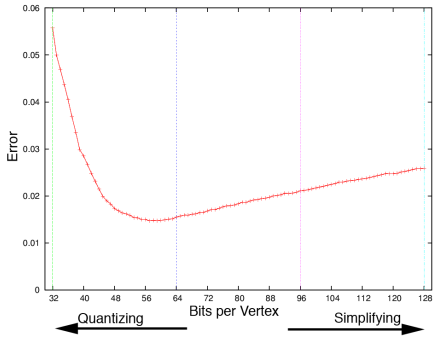


Figure 11: *Computing the optimal bits per vertex of a given target size of 136KB for the bunny model.*

Model	# Verts (x 10^3)	Orig.Size (KB)	Simp. # Verts x10^3	Compressed Size (KB)	Vertex Attributes									Error
					x	y	z	nx	ny	nz	r	g	b	
Bunny	35	816	17	136	13	14	13	8	8	8				0.015
Dragon	438	10257	219	1710	14	14	12	8	8	8				0.022
Happy	544	12742	272	2124	13	14	14	8	7	8				0.025
Cuneiform	539	14751	315	2459	11	11	10	6	7	7	4	4	4	0.030
Thai Statue	5000	117187	1656	19408	21	22	21	11	10	11				0.001

Figure 12: *Combined quantization plus simplification for several models with a target size of 1/6 of the original model size.*

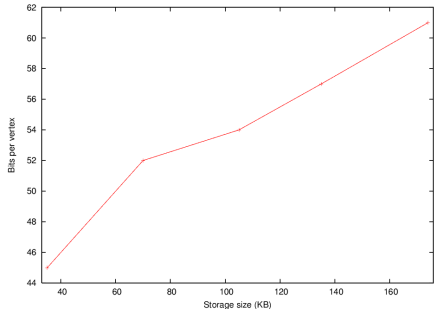


Figure 13: *Shows optimal bits per vertex on various sizes for the bunny model.*

For Figure 13, we performed an experiment to see how the best bits per vertex changes as a function of the target storage size.

We show in Figure 12 the quantitative results of applying our technique to several models, reducing each one to 1/6 of its original storage size. It is interesting to notice that there are similarities in bit allocations across models with the same set of attribute groups. Although each bit allocated differently is a significant change (each bit naturally doubles the resolution), it seems there is some hope to develop reasonable heuristics for bit allocation for some common classes of models.

6. Decompression in a Vertex Program

The ideal environment for sending packed vertex data to the video card would be to send a single array of bytes; however, restrictions from the graphics hardware drivers prevent this. The conventional method of sending vertex data to the rendering pipeline is to send a fixed-size array of attribute-dependent data types. While this allows the pipeline to be better optimized, it is less useful for sending general data to the card.

Although current vertex processing units operate only on floating-point data types, OpenGL provides a method for defining generic attributes which allows data to be sent to the card as nearly any OpenGL data type, including unsigned shorts and unsigned bytes. Using a combination of these types allows us to effectively emulate that single array of bytes (i.e. the required number of bits padded up to the nearest byte or word boundary). Using this method, data is stored in these integer types and cast into floats by the hardware before they are transferred to the vertex processor's registers. This method allows the finest level of granularity in choice of total bytes to send for each vertex, and easily allows every bit of a byte to be utilized. However, we have had mixed results in the level of optimization provided by the driver for these integer types in the context of Vertex Buffer Objects (VBO). Fortunately, things seem to be improving for both NVIDIA and ATI drivers. For ATI, in particular, we can now get fast performance for these integer types using VBO that matches the performance of floating point data.

6.1. Decompression

The vertex program decompresses the data by performing an unpacking process. The program receives a group of floats, p_i, that represent the stream of bits, S, that the vertex attributes have been packed into. These p_i arrive in the vectorized input registers as determined by the bins selected in Section 4.3. We can think of these floats as containing integral values of 16 bits or less. In addition the vertex program uses a *blueprint* of the data layout that includes a mapping of attributes to bins and the predigested values, rs_i and ls_i, that are used to extract the correct bits of each attribute from the correct p_i.

As GPUs mature we expect that future cards will be able to perform integer operations natively, and the extraction will be achieved with a simple bit mask and right shift. However, on modern graphics cards, extracting a portion of bits of a particular attribute (the target bits) from a particular p_i is achieved in four steps:

1. **Right Shift:** Multiply p_i by rs_i. The value of rs_i is precomputed so that this operation effectively shifts the target bits to the immediate right of the decimal point.
2. **Frac:** The *frac* operator returns the fractional portion of a number. Performing a *frac* on the previous result effectively removes the bits to the left of the target bits.
3. **Left Shift:** Multiply the previous result by ls_i. The value of ls_i is precomputed so that this operation effectively shifts the target bits so that they are the correct significance when combined with bits from another p_i.
4. **Floor:** Perform a floor on the previous results. This step effectively removes any bits to the right of the target bits. It may be skipped for the rightmost attribute in a vector element.

We propose two methods of assembling the unpacked target bits from above into the final attributes: a general method that is compatible with all layouts respecting the bin constraints, and a customizing method that optimizes for a particular layout.

6.1.1. General Decompression

The general decompression uses a single vertex program that is compatible with any layout. Several uniform parameters to the program specify how the unpacking should be performed. For each attribute we sum the results of the application of the above steps on each of the p_i in a target bin. Each attribute is constrained to be entirely in a single bin so that this can be vectorized on the graphics hardware. The following is a sample of Cg code that will unpack the x attribute:

```
results = floor(lsx*frac(rsx*current_bin));
results.xy = results.xy+results.zw;
pos.x = results.x+results.y;
```

The entire unpacking routine requires 6 instructions for each attribute plus an addition overhead of computing which bin each attribute uses. Taking each factor into account, the unpacking process adds $5a + ba + 2b - 2$ instructions to a vertex program where a is the number of attributes and b is the number of bins.

6.1.2. Customized Decompression

Notice that the value of many of the components of `results` from the previous section will simply be zero because the current attribute has no bits in that particular p_i. We can further optimize by writing a custom vertex program for a given layout that utilizes each p_i. This custom program is generated during program initialization by analyzing the decompression blueprint. Instead of operating on one attribute at a time, we instead operate on up to four attributes at a time. Because we are customizing the program for a particular layout, we know which attribute will accumulate each component of `results`. We can also relax the constraint that each attribute be entirely in a single bin because we do not have to unpack whole attributes at a time. The following is a sample of Cg code that will unpack several attributes in a single extraction pass:

```
results = floor(ls1*frac(rs1*bin1.xxyy));
pos.x  += results.x;
pos.y  += results.y;
pos.y  += results.z;
pos.z  += results.w;
results = floor(ls2*frac(rs2*bin1.zzww));
pos.z  += results.x;
norm.x += results.y;
norm.y += results.z;
norm.z += results.w;
```

In the above example we unpack the position and normal attribute groups in only 16 instructions, a significant savings over the general purpose routine. There is also room for further optimization, such as elimination of the second call to `floor()`. However, this approach may be less conducive to acceleration by adding special-purpose hardware.

6.2. Timing Results

We have implemented the decompression vertex program using NVIDIA's Cg 1.3 compiler and arbvp1 vertex shader profile. We have run it on both NVIDIA and ATI hardware.

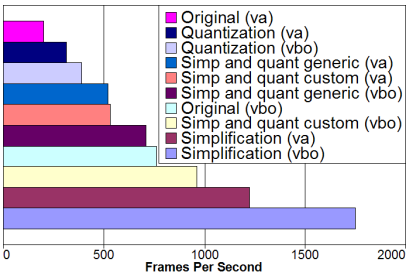


Figure 14: *Performance in frame per second of the bunny model in various configurations. The compressed versions use a target storage size of 136KB and 64 bits per vertex.*

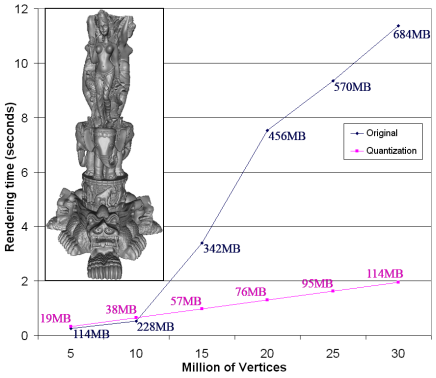


Figure 15: *Performance using VBO for one or more copies of the Thai Statue model (original model versus compressed to 1/6 the size using quantization only) on a machine with 256 MB video memory. Combining simplification makes the speedup more dramatic.*

In Figure 14, we show the performance of our algorithm running on an ATI RADEON 9800 PRO graphics card with 256 MB video memory, using a 3.06 Intel Xeon processor under the Window XP SP2 operating system. We show four different variants of the bunny model, with different tradeoffs of size and quality, rendered using either standard vertex arrays (VA) or vertex buffer object (VBO). We see that whether both models are in main memory (VA) or in video memory (VBO), our model with quantization plus simplification provides a fast, high quality alternative to the original model at one sixth the size. There is a significant speedup when the custom vertex program is used. Note while the simplification only gives the best performance, its image quality is much less than the combined version as shown in Figure 9 and Figure 10.

This data provides enough information to reason intelligently about the expected performance of compressed models for a variety of rendering scenarios. If all the models to be rendered (including textures) fit into video memory without compression, then there is no need for quantization (which could actually slow down performance). However, as the complexity of the 3D scene grows, compression provides an actual speedup, as shown in Figure 15. Another important

case where the benefits of geometry compression becomes apparent is when video memory must be shared by both texture and geometry data, effectively reducing the size of video memory for the geometry cache.

7. Proposed API Support

We envision a few important advantages of incorporating support for binned quantization directly into the graphics API:

- Ease of use: Incorporating compression into the API will simplify the user's data management both in the CPU program and the vertex program.
- Hardware support: Hiding the internals of the management of quantized data provides opportunities for optimization, including the application of additional hardware assistance.

The extended API would have a mechanism for the application to specify the mapping of attribute bits into quantization bins. When the application issues its vertex arrays, they are quantized and mapped in the specified way. Before rendering, they should be mapped back from the quantization bins to the appropriate floating point registers in preparation for execution of the application's bound vertex program. This has the flavor of the transformations currently allowed for pixel data, which may be transformed at load time from an external application format to a specific internal format.

For OpenGL, the calls to specify the mapping might look something like this:

```
glQuantizationMap( source_array,
                   attribute_index,
                   target_bin,
                   source_start_bit,
                   target_start_bit,
                   bit_length )
```

where source array is the vertex array containing application data (e.g. `GL_VERTEX_ARRAY`, `GL_NORMAL_ARRAY`, etc. or perhaps a more general `VERTEX_ATTRIB_ARRAY` index), attribute index is the vector index (0-3), target bin is the quantization bin (e.g. `GL_QUANTBIN0` might map to the same data channel as `GL_VERTEX_ATTRIB0`, `GL_QUANTBIN1` might map to `GL_VERTEX_ATTRIB1`, etc.), source start bit refers to a bit number of the quantized source, length describes how many bits are mapped, and target start bit describes where to place the bits in the target bin. Of course, there would also be an associated enum for `glEnable()` to activate the quantization feature.

The information provided by the total set of calls to `glQuantizationMap()` by the application provides enough information for the driver to infer how many bits to quantize each attribute to and what bin(s) to store them in. Actually applying the quantization to the data is a relatively fast and straightforward operation for the driver to perform, especially if the application program is using the VBO interface to inform the driver when the data is static.

The biggest potential advantage of providing driver support is the opportunity to optimize the decompression. The driver could certainly provide a vertex program similar to ours to be executed before the application's bound vertex program (perhaps with some custom microcode optimization). Even better would be to add a small unpacking stage on a separate hardware unit before the vertex processor. This would allow pipelining of the decompression with the vertex processing. This unit would ideally comprise some integer registers to perform the bitwise unpacking instructions, and possibly to allow the packed data to be passed as full 32-bit integers. If such a unit is capable of unpacking attributes spread over multiple bins without significant performance cost, that will provide the quantization optimizer processor the opportunity to create even higher quality data quantizations. The API call we have specified is sufficiently general to allow individual attributes to be spread over multiple bins, although each hardware vendor could choose additional semantic restrictions imposed by their implementation.

8. Conclusion

We have proposed and demonstrated a decompressor for compressed vertex data on current graphics hardware. The decompressor is implemented as a vertex program which accepts packed, quantized vertex attributes as input. This flexible program allows bits to be allocated arbitrarily across the set of attributes (up to 24 bits per attribute on current hardware).

In support of this decompressor, we have developed an automatic quantization algorithm using an image-space error metric. The algorithm effectively allocates appropriate bit sizes to attributes across a range of different types of attribute groups. In addition, we combine quantization compression with simplification to achieve a good balance of number of vertices to number of bits per vertex.

Our approach to vertex compression and decompression is well matched to the capabilities and needs of current graphics hardware, including the needs for independent vertex processing and binned attribute delivery. Tests have indicated that in situations where data size limits performance, our decompression is fast enough that our compression indeed yields performance speedups. Using our approach, we can achieve quality comparable to rendering of floating point data and to rendering of standard API-supported quantization, both with significant reductions in data size.

Given the increasing quantities of data being pushed through the graphics pipeline, compression is an essential tool. Even with the advent of PCI-Express, bandwidth can be a serious bottleneck requiring optimization. We have demonstrated a flexible approach to quantization-based compression that fits into the pipeline, and which could move into the graphics API itself with only modest changes.

9. Acknowledgments

We would like to thank the NVIDIA Corporation, Stanford University Graphics Lab, and National Research Council of

Canada for use of the 3D models in our demonstrations. This work has been sponsored in part by NSF Medium ITR IIS-0205586, and a DOE Early Career Award.

References

[AD01] ALLIEZ P., DESBRUN M.: Progressive compression for lossless transmission of triangle meshes. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001), Fiume E., (Ed.), Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 195–202. 2

[Cal02] CALVER D.: Vertex decompression in a shader. In *ShaderX: Vertex and Pixel Shader Tips and Tricks* (2002), Engel W. F., (Ed.), pp. 172–187. 2

[Cal04] CALVER D.: Using vertex shaders for geometry compression. In *ShaderX2: Shader Programming Tips & Tricks with DX9* (2004), Engel W. F., (Ed.), pp. 3–12. 2

[Cho97] CHOW M. M.: Optimized geometry compression for real-time rendering. In *IEEE Visualization '97 (VIS '97)* (Washington - Brussels - Tokyo, Oct. 1997), IEEE, pp. 347–354. 2

[Dee95] DEERING M.: Geometry compression. *Proceedings of SIGGRAPH'95* (1995), 13–20. 2

[GH97] GARLAND M., HECKBERT P.: Surface simplification using quadric error metrics. *Proceedings of SIGGRAPH'97* (1997), 209–215. 6

[GH98] GARLAND M., HECKBERT P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98 (VIS '98)* (Washington - Brussels - Tokyo, Oct. 1998), IEEE, pp. 263–270. 3

[Hop99] HOPPE H. H.: New quadric metric for simplifying meshes with appearance attributes. In *IEEE Visualization '99* (San Francisco, 1999), Ebert D., Gross M.,, Hamann B., (Eds.), IEEE, pp. 59–66. 3

[HV01] HAO X., VARSHNEY A.: Variable-precision rendering. In *Symposium on Interactive 3D Graphics* (2001), pp. 149–158. 2

[Ise01] ISENBURG M.: Triangle strip compression. In *Computer Graphics Forum*, Duke D., Scopigno R., (Eds.), vol. 20(2). Blackwell Publishing, 2001, ch. 91-101. 2

[Ise02] ISENBURG M.: Compressing polygon mesh connectivity with degree duality prediction. In *Proceedings of Graphics Interface* (May 2002), pp. 161–170. 2

[KG00] KARNI Z., GOTSMAN C.: Spectral compression of mesh geometry. In *Proceedings of the 16th European Workshop on Computational Geometry (EUROCG-00)* (Mar. 13–15 2000), pp. 27–30. 2

[KG02] KRONROD B., GOTSMAN C.: Optimized compression of triangle mesh geometry using prediction trees. In *Proceedings of the 1st International Symposium on 3D Data Processing Visualization and Transmission (3DPVT-02)* (June 29–21 2002), pp. 602–608. 2

[KR99] KING D., ROSSIGNAC J.: Optimal bit allocation in compressed 3d models. *Journal of Comp. Geom., Theory and Applications 14* (Nov. 1999), 91–118. 2

[KSS00] KHODAKOVSKY A., SCHRÖDER P., SWELDENS W.: Progressive geometry compression. In *Siggraph 2000, Computer Graphics Proceedings* (2000), pp. 271–278. 2

[Lin00] LINDSTROM P.: *Model Simplification Using Image and Geometry-Based Metrics*. Phd dissertation, Georgia Institute of Technology, 2000. 4

[LRC*02] LUEBKE D., REDDY M., COHEN J., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2002. 6

[LT00] LINDSTROM P., TURK G.: Image-driven simplification. In *ACM Transactions on Graphics* (2000), vol. 19 (3), pp. 204–241. 2, 3

[PR00] PAJAROLA R., ROSSIGNAC J.: Compressed progressive meshes. In *IEEE Transactions on Visualization and Computer Graphics*, Hagen H., (Ed.), vol. 6 (1). 2000, pp. 79–93. 2

[SKR01] SZYMCZAK, KING, ROSSIGNAC: An edgebreaker-based efficient compression scheme for regular meshes. *CGTA: Computational Geometry: Theory and Applications 20* (2001). 2

[Tau95] TAUBIN G.: A signal processing approach to fair surface design. *Computer Graphics 29*, Annual Conference Series (Nov. 1995), 351–358. 2

[TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Graphics Interface* (June 1998), pp. 26–34. 2

[TR98] TAUBIN G., ROSSIGNAC J.: Geometric compression through topological surgery. *ACM Transactions on Graphics 17*, 2 (1998), 84–115. 2