Encoding Normal Vectors using Optimized Spherical Coordinates

J. Smith, G. Petrova, S. Schaefer

Texas A&M University, USA

Abstract

We present a method for encoding unit vectors based on spherical coordinates that out-performs existing encoding methods both in terms of accuracy and encoding/decoding time. Given a tolerance ϵ, we solve a simple, discrete optimization problem to find a set of points on the unit sphere that can trivially be indexed such that the difference in angle between the encoded vector and the original are no more than ϵ apart. To encode a unit vector, we simply compute its spherical coordinates and round the result based on the prior optimization solution. We also present a moving frame method that further reduces the amount of data to be encoded when vectors have some coherence. Our method is extremely fast in terms of encoding and decoding both of which take constant time $O(1)$. The accuracy of our encoding is also comparable or better than previous methods for encoding unit vectors.

1. Introduction

Recent advances of technology and computational power call for novel approaches to handling large data sets in terms of their transmission, storage and processing. In Computer Graphics, large data sets arise in a variety of applications. For example, 3D scanners such as those used in the Digital Michelangelo Project [1] produce on the order of hundreds of millions of point samples per statue. LIght-Detection-And-Ranging (LIDAR) typically uses laser range scanners to scan large terrain areas and can produce billions to tens of billions of samples per scan. Rendering methods such as Photon Mapping [2] generate millions of photons. Furthermore, as computation and storage have become cheaper, larger, more complex surfaces are becoming common. For example, hierarchical modeling tools such as ZBrush can easily produce surfaces with millions of polygons.

These large data sets require efficient techniques for transmission and storage. While there are many different types of data to compress, we focus on 3D unit vectors. Such vectors appear in many applications in Computer Graphics. For example, these vectors are used to represent normals on surfaces, to modify lighting equations when used in normal maps or to store photon directions in photon maps. Unit vectors can also be viewed as points on the unit sphere and, as such, have applications in Astrophysics [3].

Naïvely storing unit vectors as three 32 bit numbers (96 bits total) is wasteful. Meyer et al. [4] showed that this representation is redundant and only 51 bits are sufficient to represent unit vectors within floating point precision. However, floating point accuracy is not always necessary, and thus good encoding/decoding techniques for 3D unit vectors that bound the maximum encoding error are needed. Such methods must be accurate, robust, and computationally efficient for both encoding and decoding (since data may need to be encoded in a streaming fashion).

1.1. Related Work

Encoding/decoding of unit vectors is a well-studied topic in Computer Graphics. The problem can be reformulated as constructing a distribution of points on the unit sphere and providing a method for finding the closest point in the distribution to a given input vector.

One of the first methods for geometry compression is due to Deering [5] who encodes normal vectors by intersecting the sphere with the coordinate octants and then dividing the portion of the sphere within each octant into six equally shaped spherical triangles. Deering then uses a uniform grid restricted to a triangle and finds the closest point on the sphere to the input normal vector. Unfortunately, there is no error analysis of this encoding technique. In addition, the encoding requires finding the closest vector from a list of vectors, which has computational cost that is exponential in the number of encoded bits.

The most well-known and popular method for encoding unit vectors is based on octahedron subdivision [6, 7, 8, 9]. The method begins with an octahedron and alternates linear subdivision and projection back to the sphere to build a point distribution. The encoding procedure is simply to identify the octant of the input vector and perform local subdivision around that vector. Hence, the encoding time is linear in the number of bits used to encode the result. While the same procedure can be used to decode the vector, the more common implementation is to use a table lookup. The latter is quite fast, but as Meyer et al. [4] point out, for high levels of accuracy the lookup table can dominate the storage costs and may not even fit in memory.

Oliveira et al. [8] and Griffith et al. [9] both explore using platonic solids other than octahedra for encoding unit vectors. Griffith et al. [9] show that the octahedron does not produce good coding results compared to other solids and advocate using a sphere covering with low number of faces [10]. The authors also provide a barycentric encoding method whose computational cost is proportional to the number of faces in the covering and is independent of the number of bits used to encode the vectors. However, the maximum error is poor compared to other methods. Qsplat [11] also encodes unit vectors using a warped barycentric encoding on a cube, which has error performance similar to the barycentric encoding in the work by Griffith et al. [9].

Bass et al. [12] describe an encoding using overlapping cones that works well with entropy encoders [13], but the encoding time is still linear in the number of output bits. The Octahedron Normal Vector method [4] uses an octahedron to encode unit vectors and does so by flattening the octahedron into a 2D square. The authors then place a regular grid over the square and encode the vector as an index. This flattening process can be performed with a small number of conditional operations, and both encoding and decoding take constant time. Moreover, the maximum error associated with this technique is much lower than typical octahedron encoding for the same number of bits.

Healpix [3] was not introduced in Computer Graphics but in the field of Astrophysics. The method creates a point distribution on the unit sphere for which the area associated with each point from the distribution is constant. The motivation for this technique does not come from compression but from processing spherical information and performing Fourier analysis on the sphere. Hence, the authors do not provide fast encoding or decoding methods, but this technique can still be used for compression.

There are also several methods developed to specifically compress normal maps that are designed to be used for real time applications like games (see Waveren et al. [14] for a survey). These methods take advantage of the 2D structure of the texture, which is not present in arbitrary streams of normals. Furthermore, tangent space normal maps always have a positive z component [14], which simplifies compression. Hence, these normal map methods address a more specific problem than general normal compression.

3Dc [15] is a normal map compression method implemented in graphics hardware. This method discards the z-coordinate and quantizes the maximum range of x/y in a 4×4 block of normals to 3 bits per channel. While on average the method performs well, the quantization step will produce severe artifacts if x or y use the maximum range, effectively only allowing 8 possible values for the x/y components. Even though the maximum encoding error is quite large, 3Dc compression can produce significant compression gains with little loss in quality for slowly varying data such as many tangent-space normal maps. Munkberg et al. [16] present modifications to the 3Dc algorithm that improve the quality the output.

Crytek's best fit normals [17] is not specifically a normal map compression algorithm and can be used to compress arbitrary normals. The algorithm assumes 8 bits per x/y/z component as input and observes that if we only encode unit vectors, most of the 24 bit space is not used for encoding since only a small number of points are close to the boundary of the unit sphere. Therefore, given an unencoded vector, the method searches for the best possible normal within the 256^3 possible encoded values such that, when normalized, is closest to the unencoded vector. The method speeds up this computation by precomputing the exhaustive search and storing the results in a cube map for lookup. Unfortunately this modification only improves the average encoded error and not the maximal encoded error. For 24 bits, the maximal encoding error is approximately 0.16° for best fit normals whereas other methods achieve much lower error. For example, Octahedral Normal Vectors achieves a maximal error of 0.04° with 24 bits. While popular for games, best fit normals produces very poor errors for normal encoding.

Contributions

In this paper, we present a computationally efficient method for encoding and decoding 3D unit vectors. The computation time is constant and is independent of the required accuracy. In addition, our method produces the smallest encoding size for a given maximum error when compared to other compression methods. We further improve our compression rates by using a differential

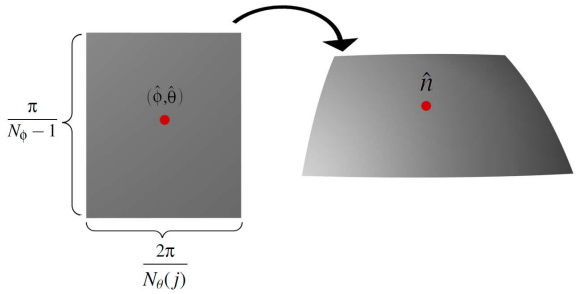

Figure 1: The encoding in Equation 2 defines a rectangular domain (left) that maps to S using spherical coordinates (right). Any point in this domain will decode to be $\hat{n}$.

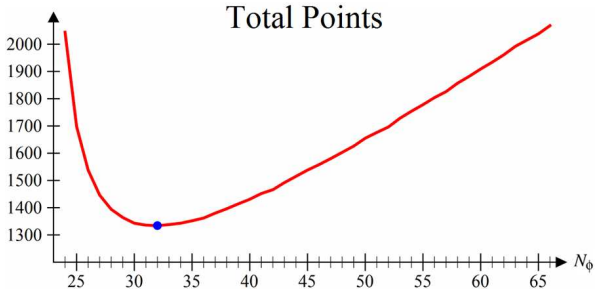

Figure 2: Total number of points generated for various values of N_ϕ with a maximum error of 4°. The minimum is 1334 points with $N_\phi = 32$.

encoding [18]. Our differential encoding uses a moving frame approach, which can be applied to any other method, and works especially well when combined with our technique.

2. Encoding

Our method is based on the spherical coordinates representation of a unit vector. Note that a naïve discretization of this representation produces poor encoding results. Instead, we use variable discretization that minimizes the number of potential encoding symbols and, hence, the encoded size.

Each point (x, y, z) on the unit sphere S has spherical coordinates $(\phi, \theta) \in [0, \pi] \times [0, 2\pi)$, where

$$x = \sin(\phi)\cos(\theta), \quad y = \sin(\phi)\sin(\theta), \quad z = \cos(\phi). \quad (1)$$

Given N_ϕ and N_θ, we consider the set $P = \{(\hat{x}, \hat{y}, \hat{z})\}$ of $N_\phi \cdot N_\theta$ points on the sphere, defined as

$$\hat{x} = \sin(\hat{\phi})\cos(\hat{\theta}), \quad y = \sin(\hat{\phi})\sin(\hat{\theta}), \quad z = \cos(\hat{\phi}),$$

where

$$(\hat{\phi}, \hat{\theta}) = \left(j\frac{\pi}{N_\phi - 1}, k\frac{2\pi}{N_\theta}\right),$$

with $j \in \{0, \ldots, N_\phi - 1\}$ and $k \in \{0, \ldots, N_\theta - 1\}$. We generate these points by dividing the parameter range for ϕ and θ into N_ϕ and N_θ uniform subintervals, respectively. Each point from P is represented by the pair (j, k). Given a unit vector n with spherical coordinates (ϕ, θ), we encode the vector by choosing a point $\hat{n} \in P$ with (j, k) determined as

$$\begin{aligned} j &= \text{round}\left(\frac{\phi(N_\phi - 1)}{\pi}\right), \\ k &= \text{round}\left(\frac{\theta N_\theta}{2\pi}\right) \bmod N_\theta, \end{aligned} \quad (2)$$

where round(x) gives the integer closest to x. Note that $\log_2(N_\phi)$ and $\log_2(N_\theta)$ bound the maximum number of bits necessary to store j and k respectively. Hence, a 96-bit floating point vector will be compressed to fewer bits if N_ϕ and N_θ are chosen appropriately.

Our goal is to select N_ϕ and N_θ in such a way that the total number of points in P is minimal for a given prescribed angle accuracy ϵ. Therefore, the angle between an encoded vector n and the corresponding decoded vector $\hat{n}$ should be $\leq \epsilon$. Since the arcs that correspond to ϕ close to 0 or π have smaller lengths than the arcs corresponding to ϕ near $\pi/2$, we can use fewer points near the poles to guarantee the desired accuracy ϵ. We achieve this effect by choosing the number N_θ adaptively, depending on j, namely $N_\theta = N_\theta(j)$. In this case, the total number of points in P will be $\sum_{j=0}^{N_\phi - 1} N_\theta(j)$.

Next, we discuss how to determine the values $N_\theta(j)$, $j = 0, \ldots, N_\phi - 1$ given a value of N_ϕ. The rounding operations in Equation 2 define a rectangular domain in terms of ϕ and θ with sides of length $\frac{\pi}{N_\phi - 1}$ and $\frac{2\pi}{N_\theta(j)}$, respectively, as shown in Figure 1. All points with coordinates (ϕ, θ) within this domain will be encoded to have the same decoded angles $(\hat{\phi}, \hat{\theta})$. Mapping this domain to the sphere creates a curved patch as shown on the right of Figure 1. Figure 4 shows a sample decomposition of the entire sphere into such patches.

Without loss of generality, we restrict ourselves to the top half of the sphere $\phi < \pi/2$. The point in the patch furthest from its center $\hat{n}$ is the bottom right (or left) corner and has spherical coordinates $(\hat{\phi} + \frac{\pi}{2(N_\phi - 1)}, \hat{\theta} + \frac{\pi}{N_\theta(j)})$. We compute the 3D vector with these coordinates using Equation 1 and take the dot product with $\hat{n}$ to yield a maximum angle of $\cos^{-1}(\cos(\hat{\phi})\cos(\hat{\phi} + \frac{\pi}{2(N_\phi - 1)}) + \cos(\frac{\pi}{N_\theta(j)})\sin(\hat{\phi})\sin(\hat{\phi} + \frac{\pi}{2(N_\phi - 1)}))$. Setting this value to be less than or equal to ϵ and solving for the smallest

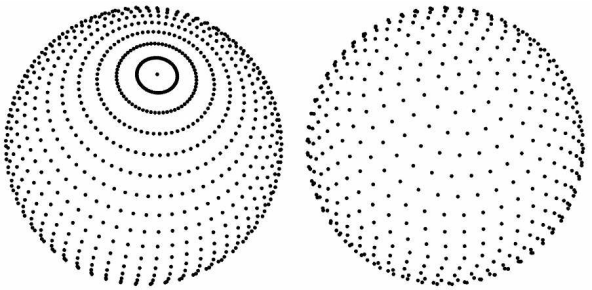

Figure 3: Point distributions on the sphere for spherical encoding using the same number of points $N_\theta(j) = 64$ for each value of j (left) and our variable number of points where $\max N_\theta(j) = 64$ (right). The spheres contain 2112 points (left) and 1334 points (right) with a maximum angle error of 4°.

integer $N_\theta(j)$ that satisfies this inequality yields

$$N_\theta(j) = \left\lceil \frac{\pi}{\cos^{-1}\left(\frac{\cos(\epsilon)-\cos(\hat{\phi})\cos(\hat{\phi}+\frac{\pi}{2(N_\phi-1)})}{\sin(\hat{\phi})\sin(\hat{\phi}+\frac{\pi}{2(N_\phi-1)})}\right)} \right\rceil.$$

Note that any value of $N_\phi \geq \frac{\pi}{2\epsilon} + 1$ yields values of $N_\theta(j)$, $j = 0, \ldots, N_\phi - 1$ such that the maximum encoding error is no more than ϵ. We need to find a value of N_ϕ for which the total number of points in P, $\sum_{j=0}^{N_\phi-1} N_\theta(j)$, attains its minimum. Figure 2 shows a graph of the total number of points in P for $\epsilon = 4°$ generated for different values of N_ϕ. When N_ϕ is close to the lower bound of $\frac{\pi}{2\epsilon} + 1$, the total number of points on the sphere is large. As N_ϕ increases, the number of points drops quickly to a minimum (in this case, 1334 points) and then increases again. To find the optimal value of N_ϕ, we simply find a neighborhood of the minimum and perform a discrete search. Notice that this optimization only has to be performed once for a value of ϵ, and the result $N_\theta(j)$ can be stored as a list of numbers and be used to encode/decode any number of vectors.

Figure 3 shows two point distributions on the sphere and demonstrates the difference between using a constant number of points for each value of N_ϕ (left) and our variable number of points (right). Each set of points will have the same maximum encoding error. However, our method is much more efficient in terms of memory. Figure 4 illustrates the regions on the sphere that our encoding in Equation 2 produces.

3. Moving Frames

Our encoding method, described in Section 2, is computationally efficient since it requires only constant time both for encoding and decoding regardless of the precision ϵ required. While our method is suitable for encoding random vectors, it has the property that for values of j close to 0 or $N_\phi - 1$, the number of possible values for k is small, as shown in Figure 4. Therefore, for unit vectors n that are close to one of the poles, we can use fewer bits to represent θ.

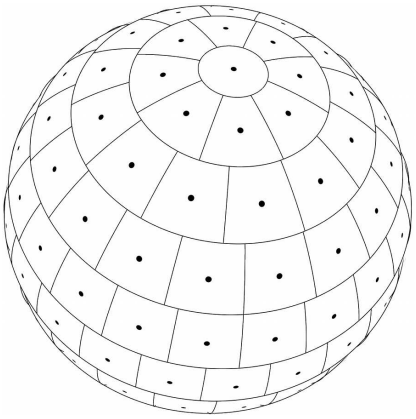

Figure 4: Our distribution of points on the sphere with a maximum error of 10° and the regions on the sphere that map to each point.

To take advantage of the above mentioned property, we will assume that we are given an ordered list of normals n^i. Let F^i be a 3×3 matrix with orthonormal columns (F^i_x, F^i_y, F^i_z) that describes the coordinate frame associated with the i^{th} vector n^i. If $\hat{n}^i = \pm F^i_z$, then we set $F^{i+1} = F^i$. If not, we define F^{i+1} as

$$\begin{aligned} F^{i+1}_z &= \hat{n}^i, \\ F^{i+1}_x &= ((F^i_z \cdot \hat{n}^i)\hat{n}^i - F^i_z)\|(F^i_z \cdot \hat{n}^i)\hat{n}^i - F^i_z\|^{-1}, \\ F^{i+1}_y &= F^{i+1}_z \times F^{i+1}_x. \end{aligned}$$

This construction builds an orthonormal frame for each normal n^{i+1} such that the z-axis aligns with the previous encoded vector $\hat{n}^i$. We then represent n^{i+1} in this coordinate frame and output the encoded values (j, k) of $(F^{i+1})^T n^{i+1}$. To decode the vector, we simply apply the decoding procedure from Section 2 and multiply by F^{i+1}, which we build from the previously decoded vector. We initialize the entire process by setting F^0 to be the Euclidean axes.

In the situation where the angle between two consecutive vectors n^i and n^{i+1} is small, this approach will produce significant compression gains because most encoded vectors will be close to the poles and use few bits to encode. Figure 5 shows the distribution of normals from the polygons of the buddha model with respect to the Euclidean axes (left) and the distribution where each

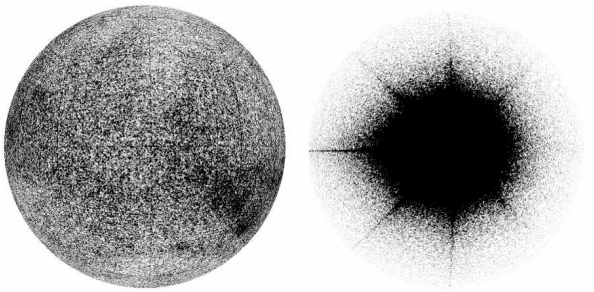

Figure 5: Normals n^i from the buddha model (left) and the normals represented in each of their coordinate frames $(F^i)^T n^i$ using our moving frame approach (right). In this figure the z-axis faces out of the page.

normal is represented in terms of its associated moving frame (right). Notice that in the first case, the normals are mostly uniformly distributed over the sphere whereas, in the second case, the normals are almost all concentrated at the positive z-axis making the silhouette of the sphere hard to see due to the lack of points away from the z-axis. The backside of the sphere on the right (not visible in this image) is extremely sparse.

This moving frame approach is not specific to our encoding method and many encoding techniques could benefit from this method, especially when coupled with an entropy encoder. However, due to the structure of our point distribution, our method is particularly well-suited to this technique.

4. Entropy Encoding

In conjunction with the moving frame, we apply an adaptive arithmetic encoder [13] to the output of our encoding method, which results in even more compression. To use this encoder, we build a distribution for j and individual distributions for k for every value of j, since a different number, $N_\theta(j)$, of symbols for k are possible for each value of j. When using the moving frame, described in Section 3, the distribution for j tends to be skewed towards zero and compresses well with the arithmetic encoder.

5. Results

We demonstrate the performance of our method by comparing it to several other methods including Healpix [3], octahedral subdivision (Octa) [6, 7, 8, 9], octahedral normal vectors (ONV) [4], sextant encoding (Sextant) [5], and the sphere1 covering (Sphere1) [9]. As test data, we use normal vectors generated from the polygons of common surfaces found in Computer Graphics. We order these vectors by performing a depth-first traversal in terms of polygon adjacency on the surface.

Figure 6 shows the graph of the encoded size (without entropy encoding or our moving frame approach) versus the maximum encoding error for the normals from the buddha model, which has 1087716 normals yielding an uncompressed size of 12747KB. Clearly, Healpix, ONV, sphere1 and our method all have better performance than Deering's sextant encoding or the popular Octa algorithm. However, our method produces a smaller encoded size than all of the other methods for a given encoding error.

Figure 7 shows the performance of all methods when we add our moving frame technique and then apply an entropy encoder [13] on the result. In all cases, entropy encoding, when combined with our moving frame, substantially reduces the encoded size. In general, Healpix, ONV and our method perform the best. However, when we decrease the maximum error and use the moving frame approach, Healpix begins to perform worse. Overall, our method performs the best and has an encoded size roughly 10% smaller than the best encoding technique. We also tested each method using entropy encoding without the moving frame but, due to the uniform distribution of points over the sphere, we did not achieve significantly better compression results than the ones from Figure 6.

While Figures 6 and 7 show encoded size versus error for a single model, Figures 8 and 9 provide more details for various models when $\epsilon \approx 1.2°$. The tables show the compressed file sizes and the encoding and decoding time in terms of seconds on an Intel Core i7 960 without our moving frame/entropy encoding (Figure 9) and with our moving frame/entropy encoding (Figure 8). Note that in Figure 8 although ONV, Healpix and our method have the same encoded size, the actual errors for these methods are 1.24°, 1.27° and 1.2° respectively. In all examples, entropy encoding/decoding dominates the running time. Our encoding/decoding is very efficient, and we consistently produce the smallest file sizes (between 14% to 45% smaller).

Healpix uses a lookup table for decoding that requires storing all possible encoded vectors in memory. In general, the decoding performance of a lookup table method is extremely fast. The drawback is that, for small maximum encoding error, these tables are quite large. While Sextant uses a constant time decoding algorithm, both Sextant and Healpix use a linear time encoding algorithm that compares each vector against all possible encoded values, which makes these methods

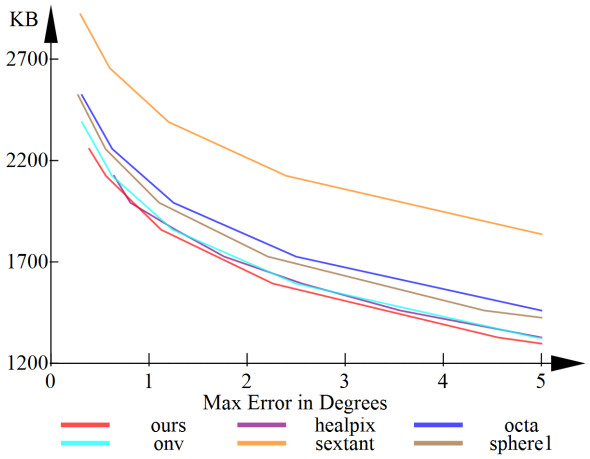

Figure 6: Compressed size without moving frames or entropy encoding versus maximum encoding error (in degrees) of various methods for the normals from the buddha model.

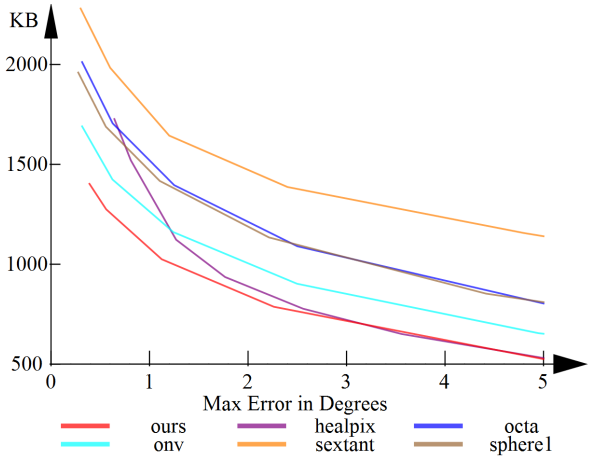

Figure 7: Compressed size using moving frames and entropy encoding versus maximum encoding error (in degrees) of various methods for the normals from the buddha model. Our moving frames technique improves the compression of all methods, though our compression benefits the most.

slow in terms of encoding time.

Some applications such as reproduction of specular highlights in graphics or engineering applications require much lower errors than 1.2°. We next perform tests for $\epsilon \approx 0.0045°$ and show the results in Figure 10 (without moving frame/entropy encoding) and Figure 11 (with moving frame/entropy encoding). Again, our method is the most efficient in terms of size using on average 23.6 bits per normal compared to the next best, ONV, at 24.6 bits per normal when using entropy encoding. At this level of error, Healpix requires a lookup table for decoding that takes more than 2GB of memory, which is not practical for most applications. Both Healpix and Sextant use linear time encoding techniques and did not complete within 24 hours. In contrast, our precomputed table of $N_\theta(j)$ takes 54 bytes for $\epsilon = 1.2°$ error and 25KB for $\epsilon = 0.0045°$ error. In terms of size, our table approximately doubles in size each time the error decreases by a factor of 0.5. This is in contrast to methods that store encoded points which quadruple in size for every decrease in ϵ by a factor of 0.5. Hence, even for very low values of ϵ, our table is still a manageable size and fits in memory.

Octa and Sphere1 use a hierarchical encoding/decoding whose complexity is proportional to the log of the number of possible encoded values. Hence, the encoding and decoding times increase significantly as ϵ decreases, especially when we remove the moving frame/entropy encoding (see Figures 8 and 10). Both ONV and our method have encoding/decoding times independent of ϵ. Despite using a few trigonometric operations, our method still outperforms ONV in terms of encoding time. However, entropy encoding/decoding takes significantly longer time for our method since we store a symbol probability distribution for each value of j.

6. Conclusions and Future Work

Our method for encoding unit vectors produces the smallest encoded size for a given error. Moreover, our encoding and decoding methods are very computationally efficient and are independent of the desired accuracy ϵ. Compared to the only other computationally efficient method, ONV, our method, both with and without our moving frame technique, consistently produces a smaller encoded size for the same error and is more efficient in terms of encoding/decoding performance. Finally, our moving frame approach significantly improves the compression results for all methods we tested, but works especially well with our encoding approach because our variable bit encoding uses fewer bits when encoded vectors are near the poles. However, the moving frame approach as well as the entropy encoding depend on the sequence of encoded normals and may not be appropriate for applications that require random access to the normals.

In the future, we would like to explore compression methods that are specialized to work with normal maps and interpolation on the GPU. When sampling data from normal maps, the GPU will bilinearly (or trilinearly if mipmapping is used) interpolate data in textures.

The result of this interpolation is typically not meaningful unless the GPU performs decompression before interpolation, as done with 3Dc compression. All of the encoding methods we have discussed suffer from this problem. The one exception is Crytek's best fit normals, which produces poor encoding results but has the property that trilinear interpolation of encoded values yields a geometrically meaningful result (although not an arc length interpolation on the sphere). Given the disparity between encoding performance of these classes of algorithms, we would like to explore building normal encoding methods that sacrifice some encoding performance but work well with the texture mapping hardware on GPUs.

Acknowledgements

This work was supported by DARPA grant HR0011-09-1-0042; the ARO/MURI grant W911NF-07-1-0185; and the NSF grant DMS 0915231.

References

[1] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, D. Fulk, The digital michelangelo project: 3d scanning of large statues, in: Proceedings of SIGGRAPH, 2000, pp. 131–144.

[2] H. W. Jensen, Realistic image synthesis using photon mapping, A. K. Peters, Ltd., 2001.

[3] K. Gorski, E. Hivon, A. Banday, B. Wandelt, F. Hansen, M. Reinecke, M. Bartelman, Healpix: A framework for high resolution discretization, and fast analysis of data distributed on the sphere, The Astrophysical Journal 622 (2) (2005) 759–771.

[4] Q. Meyer, J. Sübmuth, G. Subner, M. Stamminger, G. Greiner, On floating-point normal vectors, Computer Graphics Forum 29 (4) (2010) 1405–1409.

[5] M. Deering, Geometry compression, in: Proceedings of SIGGRAPH, 1995, pp. 13–20.

[6] G. Taubin, W. Horn, F. Lazarus, J. Rossignac, Geometry coding and vrml, Proceedings of the IEEE, Special issue on Multimedia Signal Processing 86 (6) (1998) 1228–1243.

[7] M. Botsch, A. Wiratanaya, L. Kobbelt, Efficient high quality rendering of point sampled geometry, in: Proceedings of the Eurographics workshop on Rendering, 2002, pp. 53–64.

[8] J. Oliveira, B. Buxton, Pnorms: platonic derived normals for error bound compression, in: Proceedings of the symposium on Virtual reality software and technology, 2006, pp. 324–333.

[9] E. Griffith, M. Koutek, F. Post, Fast normal vector compression with bounded error, in: Proceedings of the symposium on Geometry processing, 2007, pp. 263–272.

[10] N. Sloane, R. Hardin, W. Smith, Spherical coverings, http://www.research.att.com/ njas/coverings (1997).

[11] S. Rusinkiewicz, M. Levoy, Qsplat: a multiresolution point rendering system for large meshes, in: Proceedings of SIGGRAPH, 2000, pp. 343–352.

[12] A. Bass, K. Been, Progressive compression of normal vectors, in: Proceedings of the Symposium on 3D Data Processing, Visualization, and Transmission, 2006, pp. 1010–1017.

[13] F. Wheeler, Adaptive arithmetic coding source code, http://www.cipr.rpi.edu/˜wheeler/ac (1996).

[14] J. Waveren, I. C. no, Real-time normal map dxt compression, http://origin-developer.nvidia.com/object/ real–time–normal–map–dxt–compression.html (2008).

[15] ATI, 3dc white paper, http://www.hardwaresecrets.com/datasheets/3Dc_White_Paper.pdf (2004).

[16] J. Munkberg, T. Akenine-Möller, J. Ström, High-Quality Normal Map Compression, in: Graphics Hardware, 2006, pp. 95–102.

[17] A. Kaplanyan, Cryengine 3: Reaching the speed of light, ACM SIGGRAPH 2010 Advances in Realtime Rendering Course, http://www.crytek.com/sites/default/files/ AdvRTRend_crytek_0.ppt (2010).

[18] M. Rabbani, P. W. Jones, Digital Image Compression Techniques, SPIE, 1991.

Method	Bunny			Armadillo			Dragon			Buddha		
Ours	**246**	**.040**	.037	**591**	**.098**	.090	**1489**	**.244**	.223	**1859**	**.319**	.286
HealPix	**246**	13.786	**.023**	**591**	32.723	**.056**	**1489**	82.621	**.136**	**1859**	103.07	**.170**
Octa	263	.249	.199	633	.634	.491	1595	1.566	1.236	1992	1.957	1.519
ONV	**246**	.056	.029	**591**	.131	.065	**1489**	.330	.164	**1859**	.425	.205
Sextant	316	3.108	.024	760	7.477	.058	1914	18.859	.147	2390	23.522	.184
Sphere1	263	.322	.157	633	1.041	.378	1595	2.634	.952	1992	2.426	1.188

Figure 8: A comparison of different encoding techniques without our moving frame approach or entropy encoding at the same approximate maximum error (1.2°). For each model from left to right: encoded size in kilobytes, encoding time in seconds, decoding time in seconds. We bold the smallest size and encoding/decoding time for each model. The number of points in each model is bunny:144046, armadillo:345944, dragon:871306, buddha:1087716. The average bits per normal for each encoding technique is, Ours:14, HealPix:14, Octa:15, ONV:14, Sextant:18, Sphere1:15.

Method	Bunny			Armadillo			Dragon			Buddha		
Ours	**119**	**0.163**	**0.137**	**341**	**0.405**	**0.347**	**742**	**0.997**	**0.841**	**1003**	**1.262**	**1.067**
HealPix	137	14.691	1.777	382	35.333	4.271	840	88.982	10.770	1122	110.926	13.379
Octa	173	0.383	0.464	470	0.967	1.128	1065	2.327	2.811	1395	2.938	3.525
ONV	141	0.182	0.144	388	0.439	0.355	872	1.083	0.873	1160	1.362	1.100
Sextant	206	3.091	0.268	546	7.616	0.830	1260	18.736	1.671	1643	23.694	2.332
Sphere1	181	0.484	0.410	457	1.226	0.992	1110	2.997	2.477	1416	3.671	3.110

Figure 9: The same data as Figure 8 except we add our moving frame approach and entropy encoding to all methods. The average bits per normal for each encoding technique is, Ours:7.3, HealPix:8.3, Octa:10.4, ONV:8.5, Sextant:12.2, Sphere1:10.6.

Method	Bunny			Armadillo			Dragon			Buddha		
Ours	**528**	**.042**	.038	**1267**	**.101**	.092	**3191**	**.248**	.229	**3983**	**.309**	.293
HealPix	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Octa	580	.496	.465	1394	1.206	1.127	3509	3.013	2.825	4382	3.778	3.583
ONV	**528**	.056	**.0276**	**1267**	.137	**.074**	**3191**	.340	**.155**	**3983**	.422	**.211**
Sextant	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Sphere1	580	.711	.439	1394	1.71	1.182	3509	4.080	2.659	4382	5.348	3.365

Figure 10: A comparison of different encoding techniques without our moving frame approach or entropy encoding at the same approximate maximum error (0.0045°). The average bits per normal for each encoding technique is, Ours:30, Octa:33, ONV:30, Sphere1:33.

Method	Bunny			Armadillo			Dragon			Buddha		
Ours	**409**	.588	.626	**1031**	1.267	1.356	**2449**	2.903	3.159	**3136**	3.571	3.874
HealPix	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Octa	529	.753	1.057	1296	1.831	2.547	3206	4.569	6.425	4031	5.735	8.085
ONV	423	**.259**	**.257**	1065	**.622**	**.621**	2574	**1.592**	**1.512**	3286	**1.957**	**1.915**
Sextant	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Sphere1	503	.965	1.006	1223	2.329	2.552	3053	5.827	6.096	3834	7.271	7.656

Figure 11: The same data as in Figure 10 except we add our moving frame approach and entropy encoding to all methods. The average bits per normal for each encoding technique is, Ours:23.6, Octa:30.3, ONV:24.6, Sphere1:28.8.