Spherical Compression of Normal Vectors

Paper 1184

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.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.5]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

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 [LPC*00] 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 [Jen01] 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 polygons.

These large data sets require efficient techniques for transmission and storage. For example, even within a single computer, the bandwidth between the CPU and GPU may not be sufficient and can be a bottleneck during rendering. Therefore, compressing the data sent over that connection can lead to significant increases in rendering speed.

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 [GHB*05].

Naïvely storing unit vectors as three 32 bit numbers (96 bits total) is wasteful. Meyer et al. [MSS*10] 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. To demonstrate the point, Deering [Dee95] showed that shooting rays from the moon to Mars at floating point precision yields sub-centimeter accuracy of the final point set. Therefore, good encoding/decoding techniques for 3D unit vectors that bound the maximum encoding error are needed. Such methods must be computationally efficient for both encoding and decoding (since data may need to be encoded in a streaming fashion), accurate and robust.

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 on the required accuracy. In addition, our method produces the smallest encoding size for a given maximum error when compared to other known compression methods. We further improve our compression rates by introducing a moving frame, which can be applied to any other method, and works especially well when combined with our technique.

1.1. Related Work

Encoding/decoding 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 [Dee95] 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 [THLR98, BWK02, OB06, GKP07]. 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 [MSS*10] 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 [OB06] and Griffith et al [GKP07] both explore using platonic solids other than octahedra for encoding unit vectors. Griffith et al [GKP07] 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 [SHS97]. The authors also provide a barycentric encoding method whose computation 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 [RL00] also encodes unit vectors using a warped barycentric encoding on a cube, which has error performance similar to the barycentric encoding in Griffith et al [GKP07].

Bass et al [BB06] describe an encoding using overlapping cones that works well with entropy encoders [Whe96], but the encoding time is still linear in the number of output bits. The Octahedron Normal Vector method [MSS*10] 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.

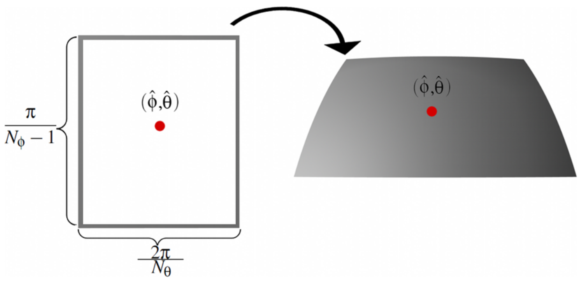

Figure 1: *The encoding in Equation 1 defines a rectangular domain (left) that maps to S using spherical coordinates (right). Any point in this domain will the central point for its encoded value.*

Healpix [GHB*05] 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 the technique can still be used for compression.

2. Encoding

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

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

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 $j \in \{0,\dots,N_\phi - 1\}$ and $k \in \{0,\dots,N_\theta - 1\}$. 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

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

where $round(x)$ gives the integer closest to x.

Our goal is to select N_ϕ and N_θ in such a way that the

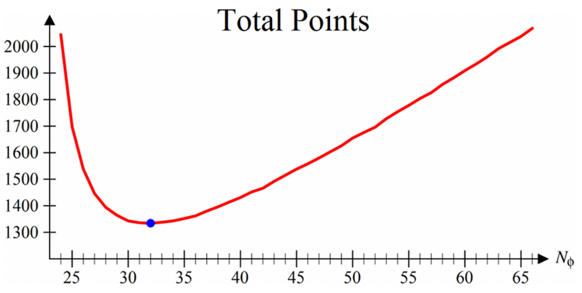

Figure 2: *Total number of points generated for various values of N_ϕ with a maximum error of 4°. The minimum is 1334 points with $N_\phi = 32$.*

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 $\leq \varepsilon$. 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 1 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.

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 $\hat{n}$ corresponds to the bottom left (or right) corner and has coordinates $(\hat{\phi} + \frac{\pi}{2(N_\phi-1)}, \hat{\theta} + \frac{2\pi}{N_\theta(j)})$. The angle between this point and $\hat{n}$ is $\cos^{-1}(\cos(\hat{\phi})\cos(\hat{\phi}+\frac{\pi}{2(N_\phi-1)}) + \cos(\frac{\pi}{N_\theta(j)})\sin(\hat{\phi})\sin(\hat{\phi}+\frac{\pi}{2(N_\phi-1)}))$. Setting this value to be less than or equal to ε and solving for the smallest integer $N_\theta(j)$ that satisfies this inequality yields

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

Note that any value of $N_\phi \geq \frac{\pi}{2\varepsilon} + 1$ 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 $\varepsilon = 4^\circ$ generated for different values of N_ϕ. When

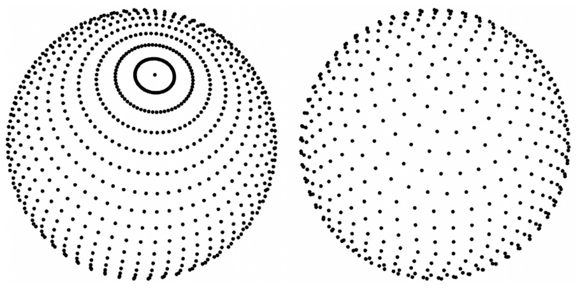

Figure 3: *Point distributions on the sphere for spherical encoding using the same number of points $N_\theta(j) = 64$ for each value of j (left) and our variable number of points where* $\max N_\theta(j) = 64$ *(right). The spheres contain 2112 points (left) and 1334 points (right) with a maximum angle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

N_ϕ is close to the lower bound of $\frac{\pi}{2\varepsilon} + 1$, 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 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 1 produces.

2.1. 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. The method is also suitable for encoding vectors without any coherence. However, we can improve the compression results if the method is applied to coherent data. In this case, we assume that we are given an ordered list of unit 3D vectors n^i that, when decoded, have value $\hat{n}^i$.

Our encoding has the property that, for values of j close to 0 or $N_\phi - 1$, the number of possible values $N_\theta(j)$ for k is small as shown in Figure 4. To take advantage of this property, we use a moving frame to encode the vectors 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 $\hat{n}^i = F^i_z$, then we set $F^{i+1} = F^i$. If not, we define F^{i+1}

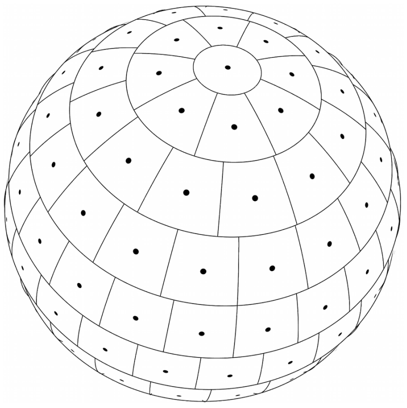

Figure 4: *Our distribution of points on the sphere with a maximum error of* 10° *and the regions on the sphere that map to each point.*

as

$$\begin{aligned} F_z^{i+1} &= \hat{n}^i, \\ F_x^{i+1} &= ((F_z^i \cdot \hat{n}^i)\hat{n}^i - F_z^i)\|(F_z^i \cdot \hat{n}^i)\hat{n}^i - F_z^i\|^{-1}, \\ F_y^{i+1} &= F_z^{i+1} \times F_x^{i+1}. \end{aligned}$$

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. 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 normal is represented in terms of its associated 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. 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.

2.2. Entropy Encoding

In conjunction with the moving frame, we apply an adaptive arithmetic encoder [Whe96] 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 k's are possible for each value of j. When using the moving frame, described in Section 2.1, the distribution for j tends to be skewed towards zero and compresses well with the arithmetic encoder.

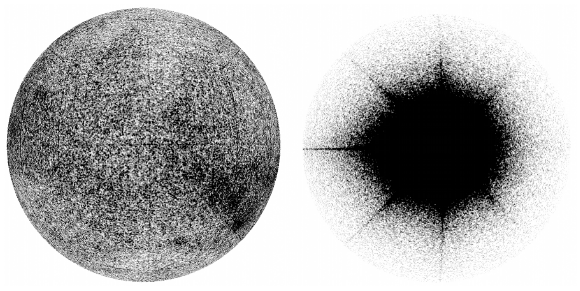

Figure 5: *Normals* n^i *from the buddha model (left) and the normals represented in each of their coordinate frames* $(F^i)^T n^i$ *using our moving frame approach (right).*

3. Results

We demonstrate the performance of our method by comparing it to several other methods including Healpix [GHB*05], octahedral subdivision (Octa) [THLR98, BWK02, OB06, GKP07], octahedral normal vectors (ONV) [MSS*10], sextant encoding (Sextant) [Dee95], and the sphere1 covering (Sphere1) [GKP07]. 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. Clearly, Healpix, ONV, sphere1 and our method all have better performance than Deering's sextant encoding or the popular Octa algorithm. However, our method and sphere1 produce a smaller encoded size than Healpix or ONV.

Figure 7 shows the performance of all methods when we add our moving frame technique and then apply the entropy encoder on the result. In all cases, entropy encoding, when combined with our moving frame, substantially reduces the encoded size. In general, Healpix, sphere1 covering and our method perform similarly in terms of error when the maximum error is greater than 2°. However, when we decrease the maximum error, sphere1 covering and especially Healpix begin to perform worse. When the maximum angle error approaches 0.5°, ONV and our method perform the best, but our method produces an encoded size roughly 15% smaller than the best encoding technique. We

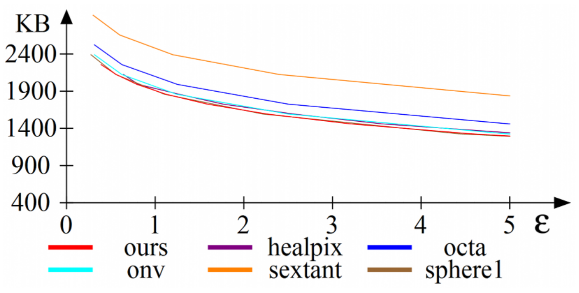

Figure 6: *Compressed size without moving frames or entropy encoding versus maximum encoding error (in degrees) of various methods for the normals from the buddha model.*

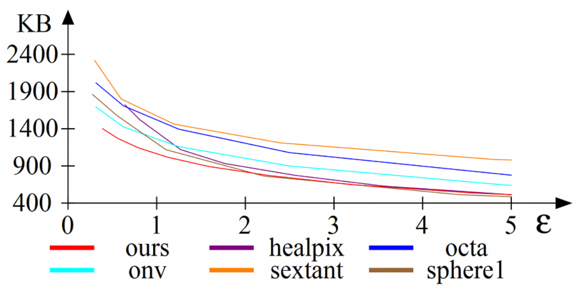

Figure 7: *Compressed size using moving frames and entropy encoding versus maximum encoding error (in degrees) of various methods for the normals from the buddha model. Our moving frames technique improves the compression of all methods, though our compression benefits the most.*

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 significant compression results over Figure 6.

Figure 8 compares our method versus other techniques on several different models where all encoding techniques have a maximum encoding error of approximately 1.2°. The compressed results in all cases use the moving frame method together with entropy encoding. The table shows the compressed file sizes and the encoding and decoding time in terms of seconds (ignoring the time used for the moving frame/entropy encoding) on an Intel Core i7 960. For all data sets, our method produces the smallest encoding size of all methods (between 10% to 33% smaller size). In all cases, our algorithm is also the fastest in terms of encoding. The encoding times for both our method and ONV are both independent of ε. Despite using a few trigonometric operations, our method still outperforms ONV encoding due to the number of conditional operations that the latter method uses. Sphere1 and Octa both take linear time in the number of encoded bits per vector. Both Healpix and Sextant encode vectors by searching through a list of points to find the point closest to the input vector. Unfortunately, the number of points on the sphere increases exponentially with the number of encoded bits per vector, which leads to significant encoding time.

In terms of decoding, Healpix, Octa, Sextant and sphere1 all use lookup methods that require storing a table of all possible encoded vectors in memory. The decoding performance of these methods is extremely fast. But, for small maximum encoding error, these tables are quite large, for example tens to hundreds of megabytes. Our decoding time is very competitive with even the fastest of these methods. Moreover, the result of our optimization $N_\theta(j)$, $j = 0,\ldots,N_\phi - 1$ takes a very small amount of space in memory. For 0.1° error, our table takes less than 1.25 kilobytes of memory.

Compared to the error efficient methods, Healpix requires substantial encoding time and our method is hundreds of times faster in terms of encoding. Both Healpix and Sphere1 require large tables of all possible encoded vectors in memory for both encoding and decoding. Moreover, in the example in Figure 8, we are approximately three times faster in terms of encoding and this gap will widen as we decrease ε. Finally, compared to 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.

4. Conclusions

Our spherical encoding method for 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. Finally, our moving frames 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.

Acknowledgements

All of the models used in the paper were obtained from the Stanford 3D Scanning Repository.

References

[BB06] Bass A., Been K.: Progressive compression of normal vectors. In *Proceedings of the Symposium on 3D Data Processing, Visualization, and Transmission* (2006), pp. 1010–1017. 2

[BWK02] Botsch M., Wiratanaya A., Kobbelt L.: Efficient high quality rendering of point sampled geometry. In *Proceedings of the Eurographics workshop on Rendering* (2002), pp. 53–64. 2, 4

[Dee95] Deering M.: Geometry compression. In *Proceedings of SIGGRAPH* (1995), pp. 13–20. 1, 2, 4

Method	Bunny			Armadillo			Dragon			Buddha		
Ours	**120**	**0.042**	0.027	**344**	**0.102**	0.066	**752**	**0.25**	0.16	**1018**	**0.323**	0.206
HealPix	132	16.552	0.025	379	40.066	0.059	837	100.25	0.147	1120	125.32	0.189
Octa	173	0.141	0.022	470	0.339	0.053	1063	0.865	0.133	1394	1.049	0.166
ONV	141	0.067	0.033	388	0.162	0.080	873	0.404	0.201	1162	0.450	0.242
Sextant	179	3.444	**0.021**	482	8.279	**0.051**	1110	20.778	**0.128**	1461	24.22	**0.159**
Sphere1	136	0.141	0.026	379	0.335	0.081	837	0.831	0.174	1119	1.01	0.226

Figure 8: *A comparison of different encoding techniques each using our moving frame approach and 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.*

[GHB*05] GORSKI K., HIVON E., BANDAY A., WANDELT B., HANSEN F., REINECKE M., BARTELMAN M.: Healpix: A framework for high resolution discretization, and fast analysis of data distributed on the sphere. *The Astrophysical Journal 622*, 2 (2005), 759–771. 1, 2, 4

[GKP07] GRIFFITH E., KOUTEK M., POST F.: Fast normal vector compression with bounded error. In *Proceedings of the symposium on Geometry processing* (2007), pp. 263–272. 2, 4

[Jen01] JENSEN H. W.: *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., 2001. 1

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The digital michelangelo project: 3d scanning of large statues. In *Proceedings of SIGGRAPH* (2000), pp. 131–144. 1

[MSS*10] MEYER Q., SÜBMUTH J., SUBNER G., STAMMINGER M., GREINER G.: On floating-point normal vectors. *Computer Graphics Forum 29*, 4 (2010), 1405–1409. 1, 2, 4

[OB06] OLIVEIRA J., BUXTON B.: Pnorms: platonic derived normals for error bound compression. In *Proceedings of the symposium on Virtual reality software and technology* (2006), pp. 324–333. 2, 4

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH* (2000), pp. 343–352. 2

[SHS97] SLOANE N., HARDIN R., SMITH W.: Spherical coverings. http://www.research.att.com/ njas/coverings, 1997. 2

[THLR98] TAUBIN G., HORN W., LAZARUS F., ROSSIGNAC J.: Geometry coding and vrml. *Proceedings of the IEEE, Special issue on Multimedia Signal Processing 86*, 6 (1998), 1228–1243. 2, 4

[Whe96] WHEELER F.: Adaptive arithmetic coding source code. http://www.cipr.rpi.edu/~wheeler/ac, 1996. 2, 4