Fast Lossy Compression of 3D Unit Vector Sets.

Sylvain Rousseau
LTCI, Telecom ParisTech,
Paris-Saclay University

Tamy Boubekeur
LTCI, Telecom ParisTech,
Paris-Saclay University

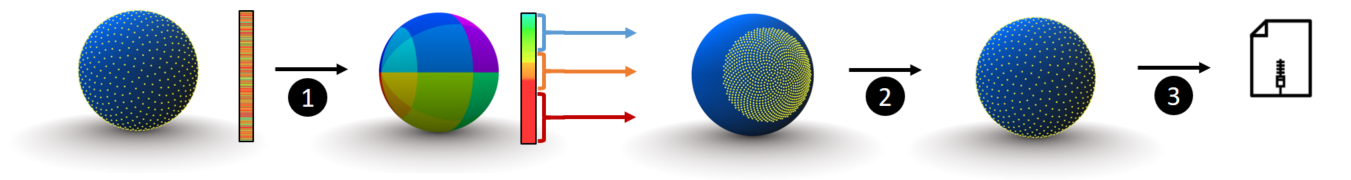

Figure 1: 1: Unit vectors are grouped according to their spherical coordinates. 2: For each group (window), we apply a *uniform mapping* that relocates subparts to the whole surface of the unit sphere. 3: To compress the vectors, we apply any existing unit vector quantization with an improved precision.

ABSTRACT

We propose a new efficient ray compression method to be used prior to transmission in a distributed Monte Carlo rendering engine. In particular, we introduce a new compression scheme for unorganized unit vector sets (ray directions) which, combined with state-of-the-art positional compression (ray origin), provides a significant gain in precision compared to classical ray quantization techniques. Given a ray set which directions lie in a subset of the Gauss sphere, our key idea consists in mapping it to the surface of the whole unit sphere, for which collaborative compression achieves a high signal-over-noise ratio. As a result, the rendering engine can distribute efficiently massive ray waves over large heterogeneous networks, with potentially low bandwidth. This opens a way toward Monte Carlo rendering in cloud computing, without the need to access specialized hardware such as rendering farms, to harvest the latent horsepower present in public institutions or companies.

CCS CONCEPTS

•Computing methodologies →Ray tracing;

KEYWORDS

Direction field compression, Unorganized normal vector set compression, Ray wave compression

ACM Reference format:
Sylvain Rousseau and Tamy Boubekeur. 2017. Fast Lossy Compression of 3D Unit Vector Sets.. In *Proceedings of SIGGRAPH Asia 2017 Technical Briefs, Bangkok, Thailand, November 27–30, 2017 (SA '17 Technical Briefs),* 4 pages.
DOI: 10.1145/3145749.3149436

1 INTRODUCTION

Over the last decade, Monte Carlo rendering has become the de facto standard in high quality visual special effects and computer animation productions [Christensen and Jarosz 2016]. This is mostly due to its physical basis, its robustness and the large number of effects accounted for by the single primitive encompassed in such methods: path tracing. In the mean time, the resolution of the generated images and the typical complexity of the input 3D scene – including geometry, materials and lighting conditions – have continuously increased. To address this ever-growing demand on computational horsepower, distributed Monte Carlo rendering engines appear as a promising solution, in particular when, beyond tile-rendering on the final image sensor, the 3D scene itself is distributed among the nodes of a computing cluster. To do so, at rendering time, the collection of rays that compose the sensor-scene-light paths are sent to each compute node to determine whether intersections occur with each node-specific region of the scene [Kato and Saito 2002; Northam et al. 2013]. As Monte Carlo path tracing provides a reasonable approximation of the rendering equation only at the price of a high amount of such rays, data i.e., ray sets exchange quickly becomes the main bottleneck of the entire image synthesis process, even with high bandwidth. Similarly, others methods such as distributed Photon mapping need to send a high amount of rays (photons) between nodes and are penalized by available bandwidth [Günther and Grosch 2014].

To solve this problem, one can use on-the-fly compression algorithms such as LZ4 [Collet 2011], a popular on-the-fly compression method for general data. This algorithm is lossless and extremely fast, but provides only a low compression ratio (up to 2). Moreover, this dictionary-based algorithm does not work with floating point data e.g., ray position and direction in our scenario. To cope with this problem, Lindstrom [2014] introduced a specific compression scheme for general floating point data, which turns out to be efficient to compress the origins of the rays, but whose generality prevents high compression ratios on the ray directions, with their particular unit nature. Indeed, several quantization methods have been developed to efficiently compress such data. Cigolle et al. [2014] provide a complete review of these methods. Since the

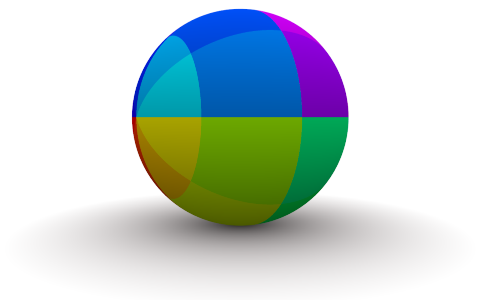

Figure 2: Unit vectors are grouped according to the most significant bits of their discretization in spherical coordinates.

publication of this survey, Keinert et al. [2015] proposed an inverse mapping for the spherical Fibonacci point set which has been used in quasi-Monte Carlo rendering algorithms [Marques et al. 2013] for the uniformity of their distribution over the surface of the sphere.

As introduced in their article, it is possible to exploit this inverse mapping to quantize ray directions in a constant time. In particular, our algorithm uses the coherence between groups of unit vectors to improve the precision of all unit vectors quantization techniques.

2 ALGORITHM

2.1 Overview

The input for our algorithm is a set of unit vectors i.e., the directions of a set of rays. In the first step (Sec. 2.2), we start by reordering them based on their Gauss sphere parametrization to form windows (groups) of spatially coherent vectors. These windows delimit subparts of the unit sphere and are mapped to the whole surface of the unit sphere using the mapping introduced in Sec. 2.3, which preserves a uniform distribution to minimize the maximal error. Then, we encode the unit vectors of each window by quantization, storing the windows with the pattern defined in Sec. 2.5. The output of the algorithm is made of both the array of compressed windows and the array containing position indices of the vectors in the original unit vector array. Since, in our target application, the order of rays is not relevant during the interactions with the scene, this second array will be kept by the master node of the cluster and used to register the returned data in the correct path.

2.2 Grouping

Our compression algorithm for unit vector sets draws inspiration from methods exploiting small windows (groups) of data to encode samples as *offsets* w.r.t. the window average. In the context of rendering, the order in which rays are sent is not relevant, which offers us an opportunity to reorganize them in a spatially coherent way to maximize window sizes. Note that some rendering engines already sort the rays during rendering [Eisenacher et al. 2013], which can be partially exploited for our grouping step. Therefore, the first step of our algorithm reorders the vector set by grouping them based on their Gauss sphere parametrization. More precisely, the grouping is done by building windows of vectors sharing the same N most significant bits in the Morton code of their discretized spherical coordinates. Figure 2 shows a color coded representation of this grouping. Here, the induced linear complexity in the number of

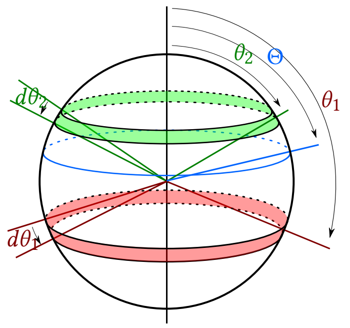

Figure 3: Notations used in Sec.2.3

vectors motivates our choice against alternatives. In particular, the clustering quality could be improved using spherical Fibonacci point sets. However, this would significantly increase compression time, while we aim at providing *on-the-fly* data compaction. Moreover, this grouping gives us an easy way to compute the average vector from any vector contained inside of the window. All vectors in a window share the same N most significant bits that we call the *key* of the window. The Morton code of the average value of the window is equal to $key + ((\sim mask)/4)$ with $mask$ being the value with its N most significant bits set to 1 and the others set to 0. With this grouping strategy, odd values of N give more uniform windows and should be favored because of step effect.

2.3 Uniform Mapping

Our on-the-fly compression scheme is entirely based on a specific quantization process. The best quantization to minimize the maximal error in the case of an unknown distribution is a uniformly distributed point set over the space in which the data is defined. The error is defined by the angle between the original and the decompressed unit vector. However, in practice, the range of directions associated with a local set of rays spans only a region of the Gauss sphere. Therefore, we propose to use a uniform mapping, from a region of the surface to the whole surface of the unit sphere.

In Fig. 3, we define the notations that we use. We also define in Eq. 1 dS_1 (resp. dS_2) the red (resp. green) surfaces in Fig. 3. In Eq. 2, we define S_{cap}, the surface of the spherical cap i.e., the region of the sphere defined by all the points that form an angle with a given vector which is smaller than a given threshold Θ. In the following, we refer to the *average unit vector* as the vector at the center of the surface of the window.

$$\begin{aligned} dS_1 &= 2\pi r^2 \sin(\theta_1) d\theta_1 \\ dS_2 &= 2\pi r^2 \sin(\theta_2) d\theta_2 \end{aligned} \tag{1}$$

$$\begin{aligned} S_{cap} &= 2 \int_{\theta \in [0,\Theta]} \pi r^2 \sin\theta d\theta \\ &= 2\pi r^2 (1 - \cos\Theta) \end{aligned} \tag{2}$$

We want to find a mapping such as:

$$\frac{dS_1}{S_{sphere}} = \frac{dS_2}{S_{cap}} \tag{3}$$

The density can be rewritten as:

$$\frac{dS_1}{S_{sphere}} = \frac{2\pi r^2 \sin\theta_1 d\theta_1}{4\pi r^2} = \frac{\sin\theta_1 d\theta_1}{2} \tag{4}$$

$$\frac{dS_2}{S_{cap}} = \frac{2\pi r^2 \sin\theta_2 d\theta_2}{2\pi r^2(1-\cos\Theta)} = \frac{\sin\theta_2 d\theta_2}{1-\cos\Theta} \tag{5}$$

We seek $h : \theta_1 \longrightarrow \theta_2$, a mapping function. Using Eq. 3:

$$\sin\theta_2 d\theta_2 = \frac{1-\cos\Theta}{2}\sin\theta_1 d\theta_1 \tag{6}$$

$$-d(\cos\theta_2) = \frac{1-\cos\Theta}{2}(-d(\cos\theta_1)) \tag{7}$$

$$\cos\theta_2 = \frac{1-\cos\Theta}{2}\cos\theta_1 + c \tag{8}$$

We define $h(0) = 0$, a unit vector equal to the average that will be mapped to itself. We obtain:

$$c = 1 - \frac{1-\cos\Theta}{2} \tag{9}$$

Then, we can compute θ_2 with θ_1:

$$\cos\theta_2 = 1 + \frac{1-\cos\Theta}{2}(\cos\theta_1 - 1) \tag{10}$$

$$\theta_2 = \operatorname{acos}\left(1 - \frac{1-\cos\Theta}{2}(1-\cos\theta_1)\right) \tag{11}$$

Eq. 11 gives us the mapping from θ_1 to θ_2 as illustrated in Fig. 3. This can be written as:

$$h(\theta_1) = \operatorname{acos}\left(1 - \frac{1-\cos\theta_1}{k}\right) \text{ with } k = \frac{2}{1-\cos\Theta} \tag{12}$$

The inverse function for decompression has the following form:

$$h^{-1}(\theta_2) = \operatorname{acos}(1 - k(1-\cos\theta_2)) \tag{13}$$

Interestingly, as we can see in Eq. 13, the exact same mapping function is used for compression and decompression, using k for compression and $\frac{1}{k}$ for decompression.

With h providing a mapping from an angle to another angle, we can apply this mapping to the unit vector directly. Let $\mathbf{x}$ be the original unit vector, $\mathbf{x'}$ the mapped one and $\mathbf{P_0}$ the normalized average vector. Let's define:

$$l_0 = \operatorname{acos} < \mathbf{P_0}, \mathbf{x} > \tag{14}$$

$$l_1 = h(l_0) \tag{15}$$

$$= \operatorname{acos}\left(1 - \frac{1- < \mathbf{P_0}, \mathbf{x} >}{k}\right) \tag{16}$$

We define $\mathbf{P_1}$, a vector orthogonal to $\mathbf{P_0}$ in the same plane as $\mathbf{x}$.

$$\mathbf{P_1} = \frac{\mathbf{x} - < \mathbf{x}, \mathbf{P_0} > \mathbf{P_0}}{||\mathbf{x} - < \mathbf{x}, \mathbf{P_0} > \mathbf{P_0}||} \tag{17}$$

Using $\mathbf{P_0}$ and $\mathbf{P_1}$ as axes, $\mathbf{x'}$ is defined as :

$$\mathbf{x'} = \cos(l_1)\mathbf{P_0} + \sin(l_1)\mathbf{P_1} \tag{18}$$

which gives us the mapping from a point $\mathbf{x}$ to a point $\mathbf{x'}$:

$$\mathbf{x'} = c\mathbf{P_0} + \sqrt{1-c^2}\mathbf{P_1} \text{ with } c = 1 - \frac{1- < \mathbf{P_0}, \mathbf{x} >}{k} \tag{19}$$

This mapping can be easily implemented (List. 1). Its use on hemispherical Fibonacci point set is shown in Fig. 4

Listing 1: C++ code for the uniform mapping

```
vec3 mapping(vec3 & x, vec3 & p0, double ratio)
{
    double k = ratio;
    double d = dot(x, p0);
    vec3 p1 = normalize(x − d ∗ p0);
    double c = (1 − ((1 − d) / k));
    return  p1 ∗ sqrt(1 − (c ∗ c)) + (c ∗ p0);
}
```

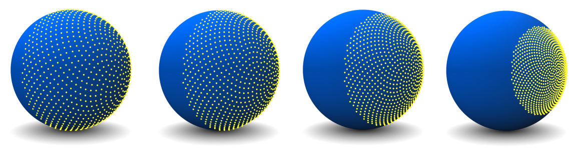
Figure 4: Mapping applied to hemispherical Fibonacci point sets. From left to right: original point set, mapping with k = 0.75, 0.5 and 0.25.

Window 1		...	Window n	
N_1	$C_{1\ 1}$ ... $C_{1\ N_1}$	...	N_n	$C_{n\ 1}$ ... $C_{n\ N_n}$

Figure 5: Windows compression pattern. N_i (resp. C_i) is the number of compressed unit vectors (resp. the compressed vectors) in the ith window.

2.4 Ratio

For a group (window) of points contained in a spherical cap S, the parameter k in the uniform mapping (Eq. 19) must be set to the value of the ratio between the length of the projection of S on the $\overrightarrow{OP_0}$ axis (with O the center of the sphere) and the diameter of the sphere. As our grouping is not uniform i.e., the shape and the area of each group are not the same, the value of the ratio is variable. For a given group with average vector P_0 and vertices of the spherical quadrilateral of the representing area B_i with $i \in 1, 2, 3, 4$, the ratio is: $\frac{1-(min(A \cdot B_i))}{2}$. This value may be under evaluated because of numerical issues, as we do not have the exact value of the average and the vertices, using a safety threshold ϵ in the implementation.

2.5 Unit vector quantization

The previous step maps a region of the unit sphere to the entire sphere. At this point, any quantization defined on the surface of the sphere can be used with an improved precision thanks to our mapping. If a minimal error is required, the state-of-the-art in uniform distribution over the surface of the sphere is the spherical Fibonacci point set. Exploiting this mapping as a quantization method can be done by using Keinert et al.'s algorithm [2015]. Because of numerical instabilities in the inverse mapping, doing computation with double precision, this approach is limited to about 8 millions of spherical Fibonacci points. If speed is the main concern, the octahedral quantization [Meyer et al. 2010] is a good tradeoff between error and performance. Using any quantization of unit vectors, we store, for each window, the number of compressed unit vectors and the quantizations associated to the mapped unit vectors. The id of the window (position in the final array) corresponds to the *key* of the group. As the average vector can be retrieved from this id (see Sec. 2.2), as well as the ratio, we do not need to store them in the window. The complete algorithm can be summarized as: (i) building windows of vectors sharing the same N most significant bits of the Morton code of their discretized spherical coordinates; (ii) for each window, storing the amount of unit vectors in the corresponding *compressed* window and applying the mapping from a subpart to the whole surface of the sphere to each vector; (iii) storing the quantization of the mapped vectors in the compressed windows. The compressed pattern is shown in Figure 5.

Table 1: The grouping is done using 13 bits. *Lin*, *oct* and *sf* are respectively the method of Lindstrom [2014], the octahedral quantization [Meyer et al. 2010] and the Spherical Fibonacci point set quantization. The number after the name of the method corresponds to the number of bits used for quantization. With *Lin*, the per-scalar rate was set to 10 and the vectors were sorted according to the Morton code of their discretized spherical coordinates prior to compression.

Method	Compression with mapping (sec)	Decompression with mapping (sec)	Mean error (°)	Max error (°)	Mean error with mapping (°)	Max error with mapping (°)	Compression Ratio with mapping (except for Lin30)
oct16	**0.42**	**0.26**	0.3370	0.9510	0.0256	0.1066	5.99019
sf16	1.39	0.48	**0.3030**	**0.5895**	0.0229	0.0696	5.99019
oct22	0.42	0.26	0.0418	0.1180	0.0031	0.0112	4.35893
sf22	1.39	0.48	0.0378	0.0700	0.0028	0.0074	4.35893
Lin30	5.249	4.483	0.0575	1.04	-	-	2.4

3 IMPLEMENTATION AND RESULTS

3.1 Implementation

We implemented our scheme in C++ using GLM and OpenMP. We benchmark it using an intel i5 6300HQ (quad core, 2.3 GHz). We use LibMorton for fast Morton code computation with the BMI2 instruction set. The option /fp:precise was activated in Microsoft Visual Studio for compilation. Due to numerical imprecision, the mapping should be done in double precision. Each test has been realized using the same 10 million uniformly distributed random unit vectors. The implementation provided by Cigolle et al. [2014] was used for the quantization methods. ZFP library was used to compare to Lindstrom's algorithm [2014]. LZ4 was compared using the official library [Collet 2011].

3.2 Speed and Error

Our error measure is the angle, in degrees, between the original vector and the compressed vector once decompressed. We evaluate the precision with the maximal and the mean error where the maximal is the largest error of the 10 million compressed vectors and the mean error is the average of these errors. As shown in Tab. 1, the method proposed by Lindstrom [2014] does not perform as well as classical quantization methods on unit vectors. This can be explained by the fact that this method was designed to handle a wide variety of floating point data, and does not use the properties of unit vectors. The lossless compression LZ4 applied to our data fails to compress as it was not designed to compress floating point data. In terms of performance, our uniform mapping (without quantization) can process 995 MB/s, and the grouping of the vectors takes 1.3 seconds. A smaller amount of windows reduces the grouping timing. Many different quantization methods exist and we choose to show the result of our mapping applied to two popular ones. The spherical Fibonacci point sets offer the best quantization with both the minimal mean and max error, while the octahedral quantization offers a good tradeoff between precision and encoding/decoding time. In terms of memory, the overhead for encoding a compressed window is 32 bits. Using a 13 bits grouping with 10 million vectors implies an overhead of 0.03 bits per encoded unit vector. If fewer vectors need to be compressed, the amount of bits used for division should be decreased. More results are provided as supplemental materials.

On shown results, our mapping reduce the mean error by a factor around 13 and the maximal error by a factor between 8.5 and 10.5. The highlighted cells show that using both Spherical Fibonacci point set or Octahedral quantization on 16 bits combined with our mapping function gives better mean and max error than the quantization without mapping using 22 bits. In this example, the use of our method increases the compression ratio from 4.36 to 5.99 with an improved accuracy.

4 CONCLUSION AND FUTURE WORK

We have presented a technique for on-the-fly compression of a set of unorganized unit vectors which, combined with state-of-the-art positional compression, can be used to compress rays in the pipeline of a distributed Monte Carlo renderer with lower compression error. The computational cost of the improved compression ratio is small in addition to classical quantization techniques and can be even reduced when used in a rendering engine that already sorts rays such as Hyperion [Eisenacher et al. 2013]. In the future, a better grouping could be designed to provide groups which shapes are close to spherical caps on the Gauss Sphere. Our algorithm could also be adapted to other kinds of data using unit vectors. e.g., compression of surfels or normal maps. We also plan to evaluate it in the context of a production renderer.

ACKNOWLEDGMENTS

This work was partially supported by the French National Research Agengy (ANR) under grant ANR 16-LCV2-0009-01 ALLEGORI, by BPI France under grant PAPAYA and by the DGA We thank Ubisoft Motion Picture for their feedback.

REFERENCES

Per H. Christensen and Wojciech Jarosz. 2016. The Path to Path-Traced Movies. *Foundations and Trends in Computer Graphics and Vision* 10, 2 (2016), 103–175.

Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. 2014. A Survey of Efficient Representations for Independent Unit Vectors. *Journal of Computer Graphics Techniques (JCGT)* 3, 2 (2014), 1–30.

Yann Collet. 2011. LZ4: Extremely Fast Compression Algorithm. (2011). http://www.lz4.org/

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted deferred shading for production path tracing. *CGF* 32, 4 (2013), 125–132.

Tobias Günther and Thorsten Grosch. 2014. Distributed Out-of-Core Stochastic Progressive Photon Mapping. In *CGF*, Vol. 33. 154–166.

Toshi Kato and Jun Saito. 2002. "Kilauea": Parallel Global Illumination Renderer. In *Proc. EGPGV*. 7–16.

Benjamin Keinert, Matthias Innmann, Michael Sänger, and Marc Stamminger. 2015. Spherical Fibonacci Mapping. *ACM ToG* 34, 6 (2015), 193:1–193:7.

Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE TVCG* 20, 12 (2014), 2674–2683.

R. Marques, C. Bouville, M. Ribardière, L. P. Santos, and K. Bouatouch. 2013. Spherical Fibonacci Point Sets for Illumination Integrals. *CGF* 32, 8 (2013), 134–143.

Quirin Meyer, Jochen Süßmuth, Gerd Sußner, Marc Stamminger, and Günther Greiner. 2010. On Floating-point Normal Vectors. In *Proc. EGSR*. 1405–1409.

L. Northam, R. Smits, K. Daudjee, and J. Istead. 2013. Ray tracing in the cloud using MapReduce. In *Proc. HPCS*. 19–26.