Quantization of Unorganized Unit-Vector Sets

Sylvain Rousseau[1] Tamy Boubekeur[2,1]
[1]LTCI, Telecom Paris, Institut Polytechnique de Paris [2]Adobe

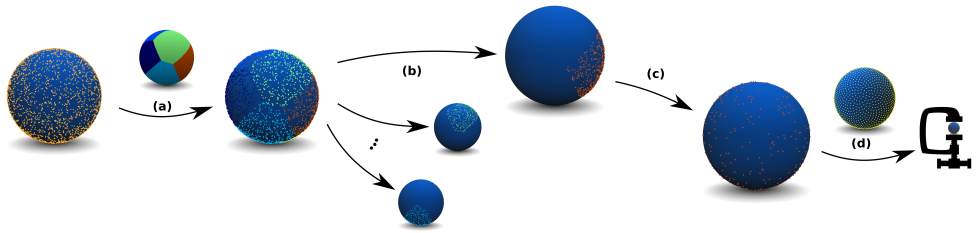


Figure 1. Unit vectors shown here as their representative points on the surface of the unit sphere are (a) grouped, then, for each group (b), called a *window*, we apply a *uniform mapping* that (c) relocates subparts to the whole surface of the unit sphere. Finally, (d), the vectors can be compressed with an improved precision by using any existing unit vector quantization.

Abstract

We present a new on-the-fly compression scheme for unorganized unit-vector sets that, using a hierarchical strategy, provides a significant gain in precision compared to classical independent unit-vector quantization methods. Given a set of unit vectors lying in a subset of the unit 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. During the compression process, the unit vectors are grouped in a way that makes them more coherent, a property that is often used in ray-tracing scenarios. The achieved compression ratio is superior to entropy-encoding methods while being easy to implement. Moreover, the constant complexity of the mapping with respect to the number of unit vectors makes our method fast. For ease of use and replication, we provide a pseudo-code along with the C++ source code. Our scheme is instrumental for applications requiring on-the-fly compression of unorganized sets of unit vectors, such as the direction of batch or wavefront of rays for rendering engines working in a distributed fashion, or the local surface orientations in an acquired 3D point cloud. The present manuscript is an extended version of a method previously published as a Technical Brief at the ACM SIGGRAPH Asia 2017 Conference.

1. Introduction

Unit vectors are a simple way to express directions and are extensively used in several fields, including astrophysics [Górski et al. 2005] and computer graphics. Recently, the significant increase in the amount of manipulated data has motivated compact representations, to ease storage, transmission, and processing. For instance, in computer graphics, unit vectors are typically used to represent the normal, measured or estimated, from a captured 3D point cloud, which may contain up to billions of samples for a single object [Levoy et al. 2000].

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]. The resolution of the generated images and the typical complexity of the input 3D scenes—including geometry, materials, and lighting conditions—have continuously increased and, today, billions of rays—embedding directions—are required to simulate light propagation when generating a single image. While the memory footprint of the rays is not necessarily an issue when working with a render farm, it becomes critical when building a rendering engine over distributed computing nodes [Kato and Saito 2002] [Somers and Wood 2012] [Northam et al. 2013] [Günther and Grosch 2014]. In such scenarios, the number of traceable rays (or photons) can quickly be bounded by the transfer performance between the nodes.

In this paper, we address this issue with a new on-the-fly method to compress sets of unorganized unit vectors, demonstrating its usage for rendering. Our method is designed to improve the data-transfer times over the network and is not suitable for local transfers i.e., GBuffer compression, in its presented form. We also make available our reference C++ implementation on GitHub under the MIT license.

1.1. Related Work

Data compression algorithms can be categorized based on different performance metrics. In our case, we target on-the-fly algorithms, meaning that the compression speed is more important than the optimal compressed size, tailored by the Kolmogorov complexity. LZ4 [Collet 2011] is a popular lossless on-the-fly compression method for general data. However, it is oblivious to the particular nature of the input data and turns out to not be very efficient, in practice, at compressing ray directions, composed of floating-point coordinates, penalizing significantly the compression ratio. To address this problem, one can look into common floating-point compression algorithms. For instance, Lindstrom [Lindstrom 2014] introduced a specific compression scheme for general floating-point data that proves to be highly effective at compressing the origins of the rays but whose generality prevents high compression ratios for the ray directions, with their particular *unit* nature. Indeed, several quantization methods have been developed to efficiently compress such unit-length data. We refer the reader to the work by Cigolle et al. [Cigolle et al. 2014] who provide a complete review of

these methods. These methods are based on one core principle: for an unknown set of unit vectors, the best compression one can achieve is to quantify them in a set of known directions. That is analogous to image compression with a fixed palette—here, the palette maps indices to directions instead of colors. Most previous methods focus on choosing a good universal direction palette, in terms of encoding and decoding performance, size, and maximization of the linear independence between directions.

Since the publication of [Cigolle et al. 2014]'s survey, Keinert et al. [Keinert et al. 2015] proposed an inverse mapping for the spherical Fibonacci point set, which is a quasi-uniformly distributed point set on the surface of the unit sphere. As introduced in their article, this inverse mapping can help quantize ray directions in constant time.

Normal mesh compression provides a good way to compress sets of normal vectors, but requires an underlying mesh connectivity. In such a case, one can use several methods based on self-similarities or high coherency between neighbors to more efficiently compress the data [Maglo et al. 2015].

Alternatively, one can use quantization methods, which are are fast and provide high compression ratio, but at the cost of a loss in data fidelity. Our approach falls in this category and brings an increased precision in the case of unorganized sets of unit vectors, instrumental for a distributed Monte Carlo rendering engine, where rays needs to be sent through a network. Our approach is generic, in the sense that a number of quantization algorithms can be injected into our scheme. We compare our method to alternatives (see Section 3), including the entropy encoding for sets of coherent independent unit vectors developed by Smith et al. [Smith et al. 2012].

2. Algorithm

2.1. Overview

The input for our algorithm is an unorganized set of unit vectors. In the first step (Section 2.2), we start by reordering them 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 Section 2.3 that 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 Section 2.5. This increased precision is due to the fact that instead of having a universal direction palette, we compress small groups of coherent unit vectors, so the mapping can create a better direction palette for each group of vectors to compress. This is similar to the image compression schemes that store deltas from the palletted values to reduce residual error. In our case, the palletted value is the ID of the windows and the offset of the mapped quantized vector. The output of the algorithm consists of both the array of compressed windows and the array containing position indices of the vectors in the original unit-vector array. We target applications in which the order

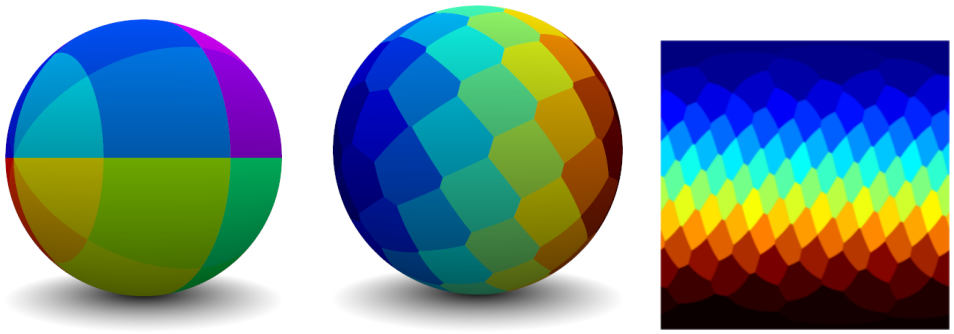
of the samples is not relevant. This is, for instance, the case in distributed rendering, where the ray's order in the original set is not relevant, i.e., each ray comes from a different light path, this array being maintained by the master node of the cluster and used to register the returned data in the correct path. If an ID is sent with the ray, this second array can even be omitted.

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 rendering context, 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 engines already sort the rays during rendering [Eisenacher et al. 2013] [Van Antwerpen 2011] [Novák et al. 2010] [Laine et al. 2013], which can be exploited for our grouping step. We present here two alternatives based on the Gauss sphere parameterization of the directions: the first method requires minimal additional memory, while the second one improves the uniformity of the groups using a lookup table.

Grouping using Discrete Spherical Coordinates (DSC)

A simple and fast way to group consists in building windows of vectors sharing the same N most significant bits in the Morton code [Morton 1966] of their normalized and discretized spherical coordinates. Figure 2 (a) shows a color-coded representation of this grouping. The linear complexity in the number of vectors, the memory used,



(a) (b) (c)

Figure 2. Unit vectors can be grouped using different techniques. We present the result of our algorithm with two different groupings: using (a) discrete spherical coordinates (DSC) which is the fastest one and (b) using the closest spherical Fibonacci point (CSF) which gives lower maximal error at the cost of a slower grouping step. The latter can be used with (c) a lookup table to improve efficiency. We denote this alternate version as CSF-LUT in the results section.

and the speed of this method make it a good candidate for the grouping step, even if the non-uniformity of the groups reduces the compression performance. Interestingly, the average vector of the windows can be easily computed. All vectors in a window share the same N most significant bits that we call the key of the window. The *average vector* can be approximated by the unit vector with its Morton code equal to $\mathrm{key} + ((\sim \mathrm{mask})/4)$, with mask being a bit field with its N most significant bits set to 1 and the others set to 0. The coordinates of this approximated average vector can be computed by (i) extracting the discretized spherical coordinates from this Morton code, (ii) un-normalizing it, and (iii) switching back to Cartesian coordinates. With this grouping strategy, odd values of N give more uniform windows and should be favored because of the stepping effect.

Grouping using Spherical Fibonacci Point Sets (CSF)

The windows are approximated in the next step as spherical caps. To reduce the maximal error, the size of the largest spherical cap should be as small as possible so we seek as uniform and as small a group as possible. A dual version of this problem is to seek a uniform distribution of the centers of the spherical caps, for which a state-of-the-art solution is given by the *spherical Fibonacci point set* that generates the positions of these centers. Therefore, we perform the grouping by assigning each unit vector to the window represented by the closest spherical Fibonacci point. We considered the inverse mapping introduced by Keinert et al. [Keinert et al. 2015], but this method alone turns out to be too slow for an on-the-fly execution. Consequently, we combine it with a precomputed lookup table such as the one shown in Figure 2 (c).

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 Figure 3, we illustrate our notation. We define in Equation (1), dS_1 (respectively, dS_2), the red (respectively, green) surfaces in Figure 3. In Equation (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 that 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 window:

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

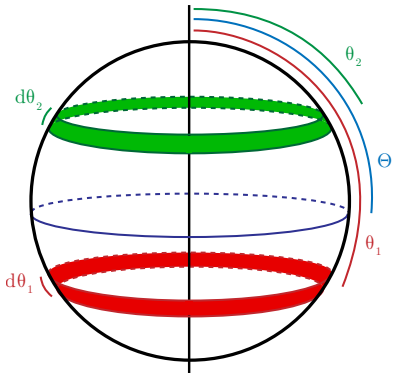


Figure 3. Notation used in Section 2.3.

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

We want to find a mapping such that

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

The density can be rewritten as

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

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

Thus, we seek a mapping function $h : \theta_1 \to \theta_2$. Using Equation (3), we find

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

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

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

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

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

We can compute θ_2 in terms of θ_1:

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

$$\theta_2 = \text{acos}\left(1 - \frac{1-\cos\Theta}{2}(1-\cos\theta_1)\right). \tag{4}$$

Equation (4) gives us the mapping from θ_1 to θ_2 as illustrated in Figure 3. This can be written as

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

The inverse function for decompression has the following form:

$$h^{-1}(\theta_2) = \text{acos}(1 - k(1-\cos\theta_2)). \tag{5}$$

Interestingly, as we can see in Equation (5), 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. We define

$$\begin{aligned} l_0 &= \text{acos} < \mathbf{P_0}, \mathbf{x} >, \\ l_1 &= h(l_0) \\ &= \text{acos}\left(1 - \frac{1- < \mathbf{P_0}, \mathbf{x} >}{k}\right). \end{aligned}$$

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

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

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

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

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

$$\mathbf{x}' = c\mathbf{P_0} + \sqrt{1-c^2}\mathbf{P_1} \text{ with } c = 1 - \frac{1- < \mathbf{P_0}, \mathbf{x} >}{k}. \tag{6}$$

This mapping is easy to implement (see Listing 1). Its use on a hemispherical Fibonacci point set is shown in Figure 4

```
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);
}
```

Listing 1. C++ code for the uniform mapping

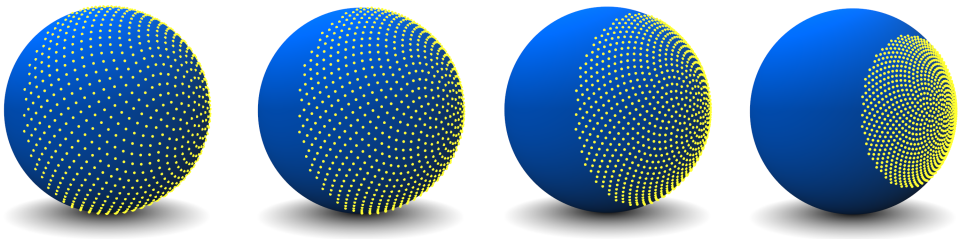


Figure 4. Mapping applied to hemispherical Fibonacci point sets. From left to right: original point set, mapping with $k = 0.75$, 0.5, and 0.25.

2.4. Ratio

For a group (window) of points contained in a spherical cap S, the parameter k in the uniform mapping (Equation (6)) must be set to the value of the ratio of the length of the projection of S on the $\overrightarrow{OP_0}$ axis (with O the center of the sphere) to the diameter of the sphere. When using DSC, the grouping is not uniform, i.e., the shape and the area of each group are not the same and 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 and quantization, which can lead to vectors located outside of this spherical cap. Therefore, for the general case, we use a safety threshold ϵ in our DSC implementation. When grouping using the spherical Fibonacci point set, we precompute k during the generation of the lookup table and use it for all the windows.

2.5. Compression Scheme

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 on a small number of bits 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 [Keinert et al. 2015]. Because of numerical instabilities in the inverse mapping, doing computation with double precision, this approach is limited to about eight million spherical Fibonacci points. If speed is the main concern, and higher precision is required, the octahedral quantization [Meyer et al. 2010] is a good tradeoff between error and performance. Given a quantization scheme, 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 Section 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 lying in subparts of the surface of the unit sphere; (ii) for each window, storing the number 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.

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 (respectively, C_i) is the number of compressed unit vectors (respectively, the compressed vectors) in the ith window.

3. Implementation and Results

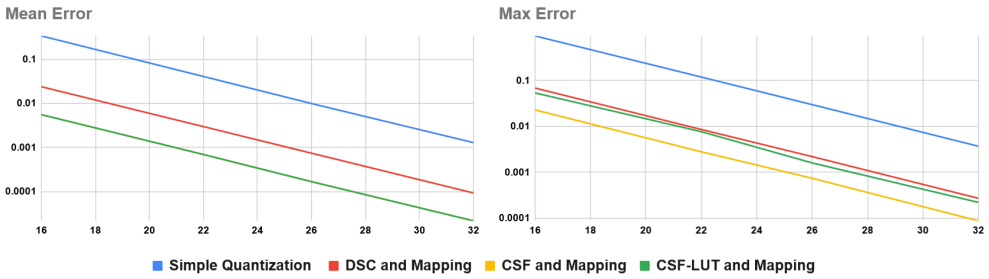
3.1. Implementation

We implemented a CPU version of our scheme in C++ using GLM and OpenMP. We benchmark it using an Intel Xeon E5-1630 v4 (quad core, 3.7 GHz). When the grouping based on spherical coordinates is used, we use LibMorton for fast Morton code computation with the BMI2 instruction set. Due to numerical imprecision, the mapping should be done in double precision. Each test has been performed using the same ten million uniformly distributed random unit vectors. The implementation provided by Cigolle et al. [Cigolle et al. 2014] was used for the octahedral quantization. The ZFP library was used for comparison to Lindstrom's algorithm [Lindstrom 2014]. Our implementation of the spherical Fibonacci quantization is provided in the supplemental material. Two application scenarios are shown in the source code—a detailed one, that could typically be used in the case of, for example, a distributed ray tracing detailed in *example.cpp*, and a simpler one, that could be used, for example, to compress the normals of a point cloud in *example_simple.cpp*.

3.2. Speed and Error

We choose to measure the error using the angle, in degrees, between the original vector and the compressed vector once decompressed, as this is the error metric used in the survey on independent unit-vector quantization by Cigolle et al. [Cigolle et al. 2014]. The shown results are computed using the octahedral quantization, but the generality of the proposed method makes it applicable to any unit-vector quantization technique. We evaluate the precision using the maximal and the mean error, where the *maximal error* is the largest compression error of the ten million compressed vectors and the *mean error* is the average of these errors.

The results that are shown in Table 1 demonstrate that the algorithm chosen for the grouping step has an important impact on the overall performance of the mapping. For example, with 2^{13} groups, a compression realized using the closest spherical Fibonacci point (CSF) for the grouping step divides the maximal error by a factor of about 3 and the mean error by a factor of about 4.3 compared to a compression realized using the discrete spherical coordinates (DSC) for the grouping step. As shown in Table 1, using the CSF grouping with 2^{13} groups divides the mean error by a factor of about 60 and the maximal error by a factor of about 42. While using a 16-bit quantization with the DSC grouping and mapping gives us performances slightly better than a classical 22-bit quantization, the CSF grouping gives slightly better performance than a 26-bit quantization as shown with the highlighted cells (respectively, in green and red) in Table 1.



Method	Simple Quantization		DSC and Mapping		CSF and Mapping		CSF-LUT and Mapping	
Error	mean	max	mean	max	mean	max	mean	max
16 bits	3.4×10^{-1}	9.5×10^{-1}	2.4×10^{-2}	6.9×10^{-2}	5.6×10^{-3}	2.3×10^{-2}	5.6×10^{-3}	5.4×10^{-2}
22 bits	4.1×10^{-2}	1.2×10^{-1}	3.0×10^{-3}	8.6×10^{-3}	7.0×10^{-4}	2.8×10^{-3}	7.0×10^{-4}	7.7×10^{-3}
26 bits	1.0×10^{-2}	3.0×10^{-2}	7.5×10^{-4}	2.2×10^{-3}	1.7×10^{-4}	7.4×10^{-4}	1.7×10^{-4}	1.6×10^{-3}
32 bits	1.3×10^{-3}	3.7×10^{-3}	9.4×10^{-5}	2.7×10^{-4}	2.2×10^{-5}	8.7×10^{-5}	2.2×10^{-5}	2.2×10^{-4}

Table 1. Quantization error (in degrees) over ten million random unit vectors, depending on the precision of the quantization and the method used for the grouping step prior to mapping. The grouping was performed using 13 bits for DSC and the equivalent number of clusters for CSF/CSF-LUT; CSF-LUT corresponds to the CSF mapping used jointly with a 500×500 lookup table.

Method	Without Mapping		With Mapping	
	mean	max	mean	max
Octahedral (16 bits)	3.4×10^{-1}	9.5×10^{-1}	2.4×10^{-2}	6.9×10^{-2}
Spherical Fibonacci (16 bits)	3.0×10^{-1}	5.9×10^{-1}	2.1×10^{-2}	6.7×10^{-2}
Octahedral (22 bits)	4.1×10^{-2}	1.2×10^{-1}	3.0×10^{-3}	8.6×10^{-3}
Spherical Fibonacci (22 bits)	3.8×10^{-2}	7.0×10^{-2}	2.8×10^{-3}	7.2×10^{-3}

Table 2. Comparison of quantization error using a 13-bit DSC grouping and two different quantization methods.

In terms of performance, the mapping step alone processes more than 1GB of data and the grouping of the ten million unit vectors takes 1.0 second using DSC and 3.86 seconds using CSF. The CSF timing can be significantly improved using a lookup table. For instance, using a 500×500 lookup table, the grouping can be done in 1.35 seconds. Depending on the requested precision, the available computational power, and the network, the grouping choice helps optimize bit-per-second gain with respect to the network bandwidth. Note, however, that this lookup table reduces the precision of our mapping as shown in Table 1 (CSF-LUT). When applied to ten million random uniform vectors, using the CSF grouping with 8192 groups, our method can encode 30.99 MB into 19.10 MB of raw data, reducing by about 38% the space needed to store the data.

Using any grouping method, a smaller number of groups speeds up the grouping step but decreases the benefit of our mapping. Moreover, we need to encode the number of unit vectors that each window contains. Therefore, the number of groups should depend on the number of unit vectors to compress. The compression of the unit vectors relies on the grouping technique used, and while DSC or CSF-LUT grouping are performed in 0.2 seconds for the ten million unit vectors, this timing increases to 0.30 seconds when using the CSF grouping. The decompression is oblivious to the grouping and takes 0.18 seconds in this example.

In Table 2, we show that our mapping can be used with different quantization methods to give improved precision. We compared our method to *entropy encoding for multiple coherent unit vectors* [Smith et al. 2012] for octahedral quantization. Smith et al. improve the compression by 26 to 34% while, with our method, we reach 38% (2^{13} groups with CSF grouping).

4. Impact of the Compression on Monte Carlo Rendering

To study the error introduced by our improved quantization process and by direction quantization in general, when applied to all the non-shadow rays (shadow rays can be more efficiently compressed using the light target) in a Monte Carlo rendering engine, we implemented our quantization scheme in a pathtracer in order to analyze the rendering quality with respect to the different parameters of our method. We used

Figure 6. Scenes used to test the impact of the algorithm on the rendering. From left to right: *staircase*, a mainly diffuse scene, *glass* a mainly specular scene, and *living room*, with a mix of different types of materials.

quantization	16 bits		24 bits		32 bits		without
mapping	with	without	with	without	with	without	without
staircase	0.983	0.592	0.984	0.975	0.984	0.984	0.984
glass	0.838	0.566	0.872	0.808	0.873	0.873	0.873
living room	0.941	0.721	0.942	0.934	0.942	0.941	0.942

Table 3. **Comparison to ground truth.** Structural similarity between the images rendered with quantization at 512 samples per pixel and the *ground truth*. We use the octahedral quantization [Meyer et al. 2010] and the vectors are grouped using the CSF grouping with 2^{13} groups. The last column shows the result when comparing the non-compressed rendering with the ground truth.

the PBRTv3 rendering engine [Pharr et al. 2016] to render the three different scenes presented in Figure 6 going from a mostly diffuse one (*staircase*) to a mostly specular one (*glass*). We rendered images at 512 samples-per-pixel (spp) with different levels of quantization, with and without our mapping. We also rendered non-quantized (512 spp) and ground truth (65536 spp) versions. We use the structural similarity metric [Wang et al. 2004] (SSIM) to compare the renderings and provide peak signal-to-noise ratio (PSNR) comparison in the supplemental materials.

Figure 7 shows the SSIM between the quantized and non-quantized renderings, for the same spp count. We can clearly see that reducing mean and maximum quantization error produces images that closely match non-quantized rendering. We can also observe that, even with a low number of bits, our quantization scheme leads to images extremely close to non-quantized rendering for a diffuse scene like the *living room*. For scenes exhibiting multiple specular bounces, such as the *glass* one, more bits are required to obtain a reasonable result. Interestingly, as reported in Table 3, activating our quantization scheme does not necessary imply a significant difference in structural similarity when compared to ground truth.

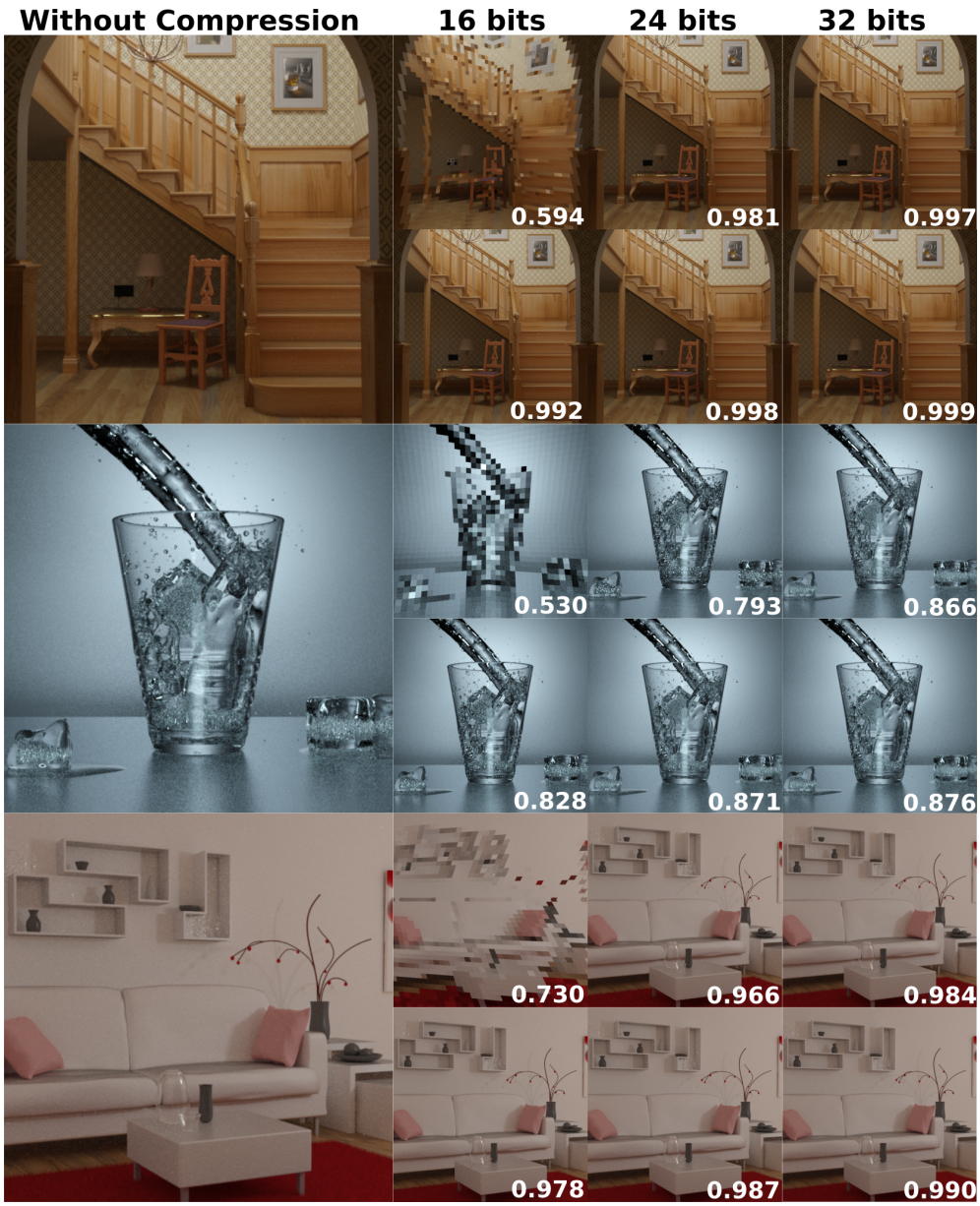


Figure 7. Quantization impact. All images were rendered using 512 spp. For each scene presented in Figure 6 (left), the first row shows quantized rendering using octahedral quantization with 16, 24, and 32 bits per direction vector, and the second row shows quantized rendering using our new mapping strategy, with 2^{13} groups and the CSF grouping, with the same bit budget. For each image we display its structural similarity (SSIM) to the non-quantized rendering output in the bottom-right corner. All quantized results are obtained by quantizing all the rays except the shadow rays.

5. Conclusion and Future Work

We have presented a new on-the-fly quantization method for unorganized sets of unit vectors which, combined with state-of-the-art positional compression, can be used to compress rays in a distributed Monte Carlo rendering framework with lower error in the final image. The extra computational cost is small in comparison to classical quantization techniques and can be even reduced when used in a rendering engine that already sorts ray batches. In the future, our method could also be used to quantize other kinds of unit-vector data sets, such as of surfels or normal maps. We showed that quantized path tracing with a high enough bit count, although different from non-quantized rendering, is not necessarily further away from ground truth. However, the resulting noise has different statistical properties, and a study on how this can affect popular denoising techniques would be interesting.

Acknowledgements

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. The scenes *staircase* and *glass* were created by blendswap user Wig42, and the *living-room* scene by the blendswap user BhaWin. All the scenes were adapted by Benedikt Bitterli for the PBRT v3 renderer [Bitterli 2016].

References

BITTERLI, B., 2016. Rendering resources. https://benedikt-bitterli.me/resources/. 34

CHRISTENSEN, P. H., AND JAROSZ, W. 2016. The path to path-traced movies. *Foundations and Trends in Computer Graphics and Vision 10*, 2, 103–175. URL: http://dx.doi.org/10.1561/0600000073. 22

CIGOLLE, Z. H., DONOW, S., EVANGELAKOS, D., MARA, M., MCGUIRE, M., AND MEYER, Q. 2014. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT) 3*, 2, 1–30. URL: http://jcgt.org/published/0003/02/01/. 22, 23, 29, 30

COLLET, Y., 2011. LZ4: Extremely fast compression algorithm. URL: http://www.lz4.org/. 22

EISENACHER, C., NICHOLS, G., SELLE, A., AND BURLEY, B. 2013. Sorted deferred shading for production path tracing. *CGF 32*, 4, 125–132. URL: https://doi.org/10.1111/cgf.12158. 24

GÓRSKI, K. M., HIVON, E., BANDAY, A. J., WANDELT, B. D., HANSEN, F. K., REINECKE, M., AND BARTELMANN, M. 2005. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *Astrophysical Journal 622* (Apr.), 759–771. arXiv:arXiv:astro-ph/0409513, doi:10.1086/427976. 22

GÜNTHER, T., AND GROSCH, T. 2014. Distributed out-of-core stochastic progressive photon mapping. *Computer Graphics Forum 33*, 6, 154–166. URL: http://dx.doi.org/10.1111/cgf.12340. 22

KATO, T., AND SAITO, J. 2002. "Kilauea": Parallel global illumination renderer. In *Proc. EGPGV*, Eurographics, Aire-la-Ville, Switzerland, 7–16. URL: https://doi.org/10.1016/S0167-8191(02)00247-8. 22

KEINERT, B., INNMANN, M., SÄNGER, M., AND STAMMINGER, M. 2015. Spherical Fibonacci mapping. *ACM ToG 34*, 6, 193:1–193:7. URL: https://doi.org/10.1145/2816795.2818131. 23, 25, 29

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels considered harmful: wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*. ACM, New York, NY, USA, 137–143. URL: https://doi.org/10.1145/2492045.2492060. 24

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The Digital Michelangelo Project: 3d scanning of large statues. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '00, 131–144. doi:10.1145/344779.344849. 22

LINDSTROM, P. 2014. Fixed-rate compressed floating-point arrays. *IEEE TVCG 20*, 12, 2674–2683. URL: https://doi.org/10.1109/TVCG.2014.2346458. 22, 29

MAGLO, A., LAVOUÉ, G., DUPONT, F., AND HUDELOT, C. 2015. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Comput. Surv. 47*, 3 (Feb.), 44:1–44:41. URL: http://doi.acm.org/10.1145/2693443, doi:10.1145/2693443. 23

MEYER, Q., SÜSSMUTH, J., SUSSNER, G., STAMMINGER, M., AND GREINER, G. 2010. On floating-point normal vectors. In *Proc. EGSR*, Eurographics, Aire-la-Ville, Switzerland, 1405–1409. URL: https://doi.org/10.1111/j.1467-8659.2010.01737.x. 29, 32

MORTON, G. M. 1966. A computer oriented geodetic data base and a new technique in file sequencing. Tech. rep., IBM Ltd, Ottawa, Canada. 24

NORTHAM, L., SMITS, R., DAUDJEE, K., AND ISTEAD, J. 2013. Ray tracing in the cloud using MapReduce. In *Proc. HPCS*. IEEE, Washington, DC, 7, 19–26. URL: https://doi.org/10.1109/HPCSim.2013.6641388. 22

NOVÁK, J., HAVRAN, V., AND DACHSBACHER, C. 2010. Path regeneration for interactive path tracing. In *Eurographics (Short Papers)*. Eurographics, Aire-la-Ville, Switzerland, 61–64. URL: http://doi.org/10.2312/egsh.20101048. 24

PHARR, M., HUMPHREYS, G., AND JAKOB, W., 2016. PBRT version 3 rendering engine. URL: https://github.com/mmp/pbrt-v3. 32

SMITH, J., PETROVA, G., AND SCHAEFER, S. 2012. Encoding normal vectors using optimized spherical coordinates. *Computers & Graphics 36*, 5, 360 – 365. Shape Modeling

International (SMI) Conference 2012. doi:https://doi.org/10.1016/j.cag.2012.03.017. 23, 31

SOMERS, B., AND WOOD, Z. J. 2012. Flexrender: A distributed rendering architecture for ray tracing huge scenes on commodity hardware. In *GRAPP/IVAPP*. IEEE, Washington, DC, 152–164. URL: https://digitalcommons.calpoly.edu/theses/812/. 22

VAN ANTWERPEN, D. 2011. Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, New York, NY, USA, 41–50. URL: https://doi.org/10.1145/2018323.2018330. 24

WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (April), 600–612. URL: https://doi.org/10.1109/TIP.2003.819861. 32

Index of Supplemental Materials

The source code is provided under the MIT license as a header only C++ library. The updated version is available on GitHub: https://github.com/superboubek/UniQuant. We provide the PSNR (psnr.svg) and SSIM (ssim.svg) values for Figure 7 at http://jcgt.org/published/0009/04/02/supplementary.zip. The rendering used to compute those values are also provided in the rendering folder there.

Author Contact Information

Sylvain Rousseau
Telecom Paris
19 Place Marguerite Perey
Palaiseau, 91120 France
sylvain@sylvainrousseau.fr
sylvainrousseau.fr

Tamy Boubekeur
Adobe
9 Rue de Milan
Paris, 75009 France
boubek@adobe.com
http://perso.telecom-paris.fr/boubek

S. Rousseau and T. Boubekeur, Quantization of Unorganized Unit-Vector Sets, *Journal of Computer Graphics Techniques (JCGT)*, vol. 9, no. 4, 21–37, 2020
http://jcgt.org/published/0009/04/02/

Received:	2019-11-04		
Recommended:	2020-01-06	Corresponding Editor:	Matt Pharr
Published:	2020-12-31	Editor-in-Chief:	Marc Olano