Permutation Coding for Vertex-Blend Attribute Compression

CHRISTOPH PETERS, Karlsruhe Institute of Technology, Germany
BASTIAN KUTH, no affiliation, Germany
QUIRIN MEYER, Coburg University of Applied Sciences and Arts, Germany

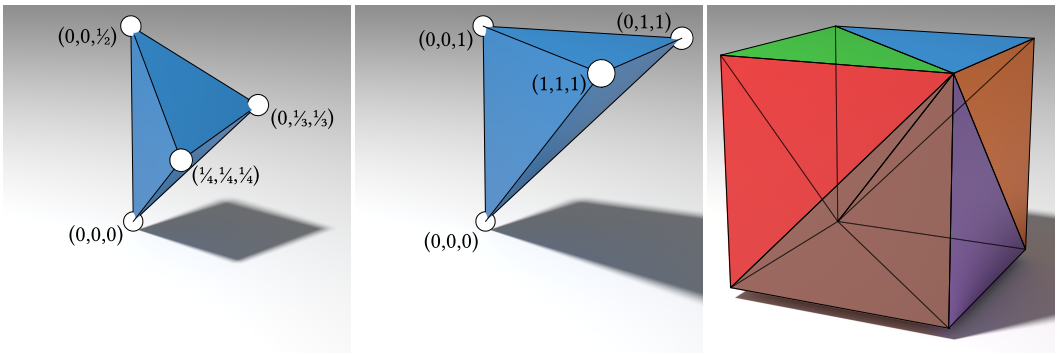

Fig. 1. Left: If we sort blend weights $0 \leq w_0 \leq \ldots \leq w_3$ with $\sum_{i=0}^{3} w_i = 1$, the vector $(w_0, w_1, w_2)^\mathsf{T}$ lies in the shown tetrahedron. Storing it coordinate by coordinate is suboptimal. Middle: We transform the tetrahedron to fill one sixth of the unit cube. Right: Based on another attribute (e.g. part of a bone index), we pick one of the 3! possible permutations of the axes (shown in different colors) and store the shuffled coordinates. During decoding, we sort the sequence to recover the weights and a permutation index from 0 to 5 that encodes the other attribute. When storing $N + 1$ weights, this strategy saves roughly $\log_2(N!)$ bits.

Compression of vertex attributes is crucial to keep bandwidth requirements in real-time rendering low. We present a method that encodes any given number of blend attributes for skinning at a fixed bit rate while keeping the worst-case error small. Our method exploits that the blend weights are sorted. With this knowledge, no information is lost when the weights get shuffled. Our permutation coding thus encodes additional data, e.g. about bone indices, into the order of the weights. We also transform the weights linearly to ensure full coverage of the representable domain. Through a thorough error analysis, we arrive at a nearly optimal quantization scheme. Our method is fast enough to decode blend attributes in a vertex shader and also to encode them at runtime, e.g. in a compute shader. Our open source implementation supports up to 13 weights in up to 64 bits.

CCS Concepts: • **Computing methodologies** → *Rendering*; **Animation**; • **Theory of computation** → **Data compression**.

Additional Key Words and Phrases: skinning, linear vertex blend animation, vertex buffer compression, vertex blend attribute compression, permutation coding, bone weights, bone indices, simplex, tetrahedron

Authors' addresses: Christoph Peters, vbac@momentsingraphics.de, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131, Karlsruhe, Germany; Bastian Kuth, kuthba86710@th-nuernberg.de, no affiliation, Germany; Quirin Meyer, quirin.meyer@hs-coburg.de, Coburg University of Applied Sciences and Arts, Friedrich-Streib-Straße 2, 96450, Coburg, Germany.

ACM Reference Format:
Christoph Peters, Bastian Kuth, and Quirin Meyer. 2022. Permutation Coding for Vertex-Blend Attribute Compression. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 1, Article 5 (May 2022), 16 pages. https://doi.org/10.1145/3522607

1 INTRODUCTION

Graphics hardware keeps improving rapidly but its computational power grows faster than the available memory bandwidth. Thus, it is increasingly important in real-time rendering to encode the scene representation as compactly as possible. This compact representation has to be usable for rendering directly. For textures, block compression accomplishes this goal. For geometry, the least intrusive approach is to compress each vertex individually at a fixed bit rate. Such methods are widely used because except for some packing code for vertex buffers and unpacking code in vertex shaders, they require no changes to the renderer. Simple fixed-point quantization works well for vertex positions and texture coordinates. For vertex normals, octahedral maps are a popular approach [Meyer et al., 2010].

Skinned meshes need additional attributes for blending: Each vertex stores multiple indices of bones and corresponding weights defining the influences of these bones. Storing only 4 influences at 8 bits per weight or index already takes 64 bits. A reasonably quantized vertex format without blend attributes takes 128 bits (see Sec. 4.3). Thus, the space requirements of blend attributes are significant. Nonetheless, their compression has not received attention until recently [Kuth and Meyer, 2021]. This first work is focused on meshes with up to four weights per vertex. The restriction to four weights is common in game engines, even though dense-weight blend skinning with many influences per vertex is known to give clear visual improvements [Le and Deng, 2013]. Kuth and Meyer [2021] also formalize naive techniques that generalize to arbitrarily many weights but these are far from optimal.

We present a more general and arguably more elegant solution. Our technique works for arbitrarily many weights and our GPU implementation supports up to 13 weights encoded in up to 64 bits per vertex. Like prior work [Kuth and Meyer, 2021], we store each tuple of bone indices only once in a table. Thus, the data stored per vertex consist of the weights themselves and a single tuple index (Sec. 3.1).

Our core insight is that no information is lost when we shuffle a strictly ordered sequence of blend weights $w_0 < \ldots < w_{N-1}$ before storage. We recover the original sequence efficiently through sorting. As we do so, we also recover the permutation that was applied to it. We then turn this permutation into an index from 0 to $N! - 1$. Since we are in control of what permutation we apply, this scheme allows us to hide $\log_2(N!)$ bits of information in the blend weights without requiring any additional storage (Sec. 3.3). In particular, we store (part of) the tuple index for the bone indices this way.

It is suboptimal to quantize the individual weights $w_0, \ldots, w_{N-1}$ directly because they are subject to inequalities. Therefore, many possible codes do not correspond to meaningful weights (Fig. 1 left). We address this issue with a linear transform that expands the space but preserves the ordering (Fig. 1 middle, Sec. 3.4). The impact of this transform on the quantization error requires a careful analysis, which also reveals a shortcoming of prior work [Kuth and Meyer, 2021] (Sec. 3.5). We overcome this shortcoming and derive nearly optimal quantization schemes for different weight counts (Sec. 3.6). In the end, all quantized numbers get coded into up to 64 bits (Sec. 3.7).

For four weights, our method is faster and more accurate than the best prior work [Kuth and Meyer, 2021]. It naturally supports more weights and scales well. We find that 48 and 64 bits provide sufficient accuracy for eight and 13 weights, respectively (Sec. 4). Our supplemental includes full source code for our renderer and our experiments.

2 RELATED WORK

The main incentives for GPU data compression are memory savings and reduced bandwidth and power consumption. Since textures typically consume most memory, GPUs provide hardware-support for random read-access from various lossy compressed texture formats [Garrard, 2020, Nystad et al., 2012]. Additionally, GPUs utilize on-the-fly compression techniques by default [McAllister et al., 2014].

Compression of mesh topology and vertex positions is well-studied but even the methods that emphasize random access usually need to decompress several faces at once [Maglo et al., 2015]. Calver [2002, 2004] introduced quantization for vertex-attribute compression by decoding vertex data in the vertex shader. Purnomo et al. [2005] carefully determine the number of bits allocated for each attribute channel. Quantization techniques are now commonly used in games [Geffroy et al., 2020, Karis et al., 2021, Persson, 2012]. Special compression schemes for unit vectors [Keinert et al., 2015, Meyer et al., 2010, Rousseau and Boubekeur, 2020] and tangent frames [Frey and Herzeg, 2011, Geffroy et al., 2020] exploit their particular properties and allow efficient decoding in vertex shaders.

Vertex blending, also known as skinning, animates a dense mesh using a hierarchy of bones [Magnenat-Thalmann et al., 1989]. In each frame, each bone holds a transformation $\mathbf{T}_i \in \mathbb{R}^{4\times4}$. Mesh vertices carry a rest position $\mathbf{p} \in \mathbb{R}^4$ (in homogeneous coordinates), indices of relevant bones $j_0, \ldots, j_N$ and corresponding weights $w_0, \ldots, w_N \geq 0$. Linear vertex blending applies the transformation of each relevant bone and computes the animated position as convex combination $\sum_{i=0}^{N} w_i \mathbf{T}_{j_i} \mathbf{p}$. It maps well to vertex shaders.

More sophisticated methods for the representation and combination of the transformations address issues such as elbow collapse, joint bulging or candy wrapper artifacts [Alexa, 2002, Kavan et al., 2008, Le and Hodgins, 2016]. When using optimized virtual bones, which blend influences of multiple bones, two weights per vertex suffice [Le and Deng, 2013]. Direct delta mush [Le and Lewis, 2019] efficiently approximates Laplacian smoothing to reduce the demands on artist-defined blend weights. It replaces scalar weights by 4×4 matrices, which can be compressed using a coarse direct delta mush and a fine vertex-blend model [Le et al., 2021]. To reduce memory demands of animations, compression of bone transformations is viable [Fréchette, 2017].

2.1 Existing Blend Attribute Compression

Thus far, there is only one work that directly addresses blend attribute compression [Kuth and Meyer, 2021]. It also formalizes some naive methods. Since we compare against all these methods, we describe them in some detail. The first step is compression of bone indices. Many vertices share exactly the same tuple of bone indices $j_0, \ldots, j_N$. Thus, the authors build a table of unique tuples and only store an index into this table per vertex (cf. Sec. 3.1).

Unit cube sampling [Kuth and Meyer, 2021] is the most naive approach for compression of blend weights. The last weight w_N is discarded since weights are known to sum to one. The remaining N weights are stored as fixed-point numbers in $[0, 1]$. Pairs of weights and bone indices can be given in any order. If the weights are sorted, we know $w_i \in [0, \frac{1}{N+1-i}]$ (cf. Sec. 3.4). The power-of-two axis-aligned bounding box (POT AABB) approach [Kuth and Meyer, 2021] rounds interval ends up to powers of two and reduces the number of bits per weight accordingly. The any AABB approach [Kuth and Meyer, 2021] quantizes different weights into integers with arbitrary range, which saves space but makes decoding more complicated (cf. Sec. 3.7).

For $N = 3$, the sorted weights are known to lie in a tetrahedron (Fig. 1 left). Thus, any encoding based on an AABB still permits lots of invalid codes. Optimal simplex sampling (OSS) [Kuth and Meyer, 2021] removes this waste. It assigns an index to each point of a regular grid that lies within

the tetrahedron and stores this index. Unlike the previous methods, OSS does not generalize to more weights naturally. Decoding requires the solution of a polynomial equation of degree N. For $N > 4$, that might be impossible in closed form [Abel, 1826].

3 OUR BLEND ATTRIBUTE COMPRESSION

Our method for blend attribute compression naturally supports an arbitrary number of weights. And as we analyze it, we identify and overcome an issue that impairs the accuracy of prior work [Kuth and Meyer, 2021]. We begin with a clear definition of the data to be stored (Sec. 3.1) and provide an overview of our method (Sec. 3.2). Then, we explain in detail how our permutation coding stores additional data alongside a sorted tuple (Sec. 3.3). To make it applicable to blend weights, we need to transform them first (Sec. 3.4). That transform necessitates a careful error analysis (Sec. 3.5). Based on the resulting insights, we derive our optimal quantization schemes (Sec. 3.6). Finally, we complete our encoding and decoding scheme (Sec. 3.7). Algorithms 3 and 4 summarize the whole procedure.

3.1 Problem Statement

Blend attributes consist of $N + 1 \in \mathbb{N}$ pairs of weights and bone indices. The order in which these pairs are given is irrelevant for the blending. We choose to sort the pairs by their weights. By convention, all blend weights are non-negative and their sum has to be exactly one. Then the blend weights $w_0, \ldots, w_N \in \mathbb{R}$ are subject to

$$0 \leq w_0 \leq w_1 \leq \ldots \leq w_{N-1} \leq w_N = 1 - \sum_{i=0}^{N-1} w_i. \tag{1}$$

Since the greatest weight w_N can be computed from the others, we do not store it explicitly.

Most of the time, nearby vertices use exactly the same bone indices. Therefore, it is inefficient to store these indices per vertex. Like prior work [Kuth and Meyer, 2021], we create a table of all tuples of bone indices in a mesh instead. Then storing the tuple of bone indices per vertex is accomplished by storing a tuple index referencing the matching entry in this table. The table would be smaller if we were to sort by bone indices instead of sorting by weights but the benefits of sorting by weights turn out to be greater (Sec. 4.1).

For creation of the table, we employ a few novel optimizations. If $w_N = 1$ after decoding, only one bone influences the vertex. In this case, the tuple index is used as bone index directly and we skip use of the table. Additionally, we exploit that bone indices for zero weights are irrelevant as we reuse tuples. Fig. 2 illustrates our strategy. We treat irrelevant indices as ∞ (or as $2^{16} - 1$ in our implementation with 16-bit indices). Then we sort lexicographically, taking the index for the largest weight as most significant. If a tuple allows reuse, it is found at the beginning of a run of matching tuples. We find all of them with a single scan over the sorted array.

Our goal is to store the blend weights $w_0, \ldots, w_N$ and the corresponding tuple index in as little memory as possible. The amount of memory should be fixed to accommodate restrictions for vertex shader inputs. The absolute worst-case error in the vector of blend weights $(w_0, \ldots, w_N)$ should be small in terms of the 2-norm. And any weight count $N + 1$ should be supported.

3.2 Overview of Our Method

As we design our method, we follow a few guiding principles that are common for fixed-rate compression: Nearly all possible codes (i.e. bit strings) should encode meaningful blend attributes. Two different codes should never encode the same blend attributes. And the worst-case error should be minimized. In particular, all weights should have roughly equal accuracy.

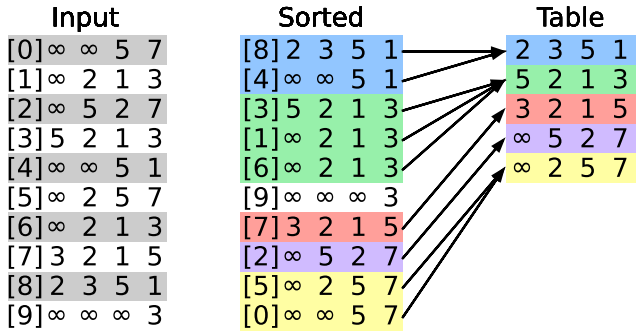

Fig. 2. Construction of the table of tuple indices. We treat irrelevant indices as ∞, sort lexicographically and scan for runs of matching tuples.

Our encoder first applies a linear transform to the vector of weights $(w_0, \ldots, w_{N-1})$ to make it fill a greater portion of the hypercube $[0,1]^N$ (Fig. 1 left and middle). This way, we avoid many invalid codes. The resulting vector gets quantized entry by entry. Due to the transform, quantizing each entry with the same precision gives lower accuracy for greater weights. Therefore, we determine suitable precision factors $B_0, \ldots, B_{N-1} \in \mathbb{N}$. Entry $i \in \{0, \ldots, N-1\}$ gets quantized into an integer $a_i B_i + b_i$ where the more significant $a_i \in \{0, \ldots, A-1\}$ provides the same precision for each entry and $b_i \in \{0, \ldots, B_i - 1\}$ provides additional precision as indicated by B_i.

We encode the tuple index and $b_0, \ldots, b_{N-1}$ into a single integer $p \in \{0, \ldots, P-1\}$, which we call the payload. All that is left to do is to store p and $a_0, \ldots, a_{N-1}$ as compactly as possible. Due to the sorted weights and the specifics of our quantization, we know $a_0 < a_1 < \ldots < a_{N-1}$. Therefore, we lose no information if we shuffle this sequence before storage using one of the $N!$ possible permutations. We use $\log_2(N!)$ bits of the payload p to pick this permutation. The remaining $\log_2 P - \log_2(N!)$ bits are stored separately (we pick the precision factors such that $P \geq N!$).

During decoding, we sort the sequence and thus recover the original $a_0, \ldots, a_{N-1}$ as well as the $\log_2(N!)$ bits of the payload. Thus, we have saved $\log_2(N!)$ bits of memory by finding a good use for all the codes, which correspond to sequences that are not sorted (Fig. 1 right).

3.3 Permutation Coding

We now describe our novel permutation coding in more detail. Its goal is to store the quantized weights $a_0 < \ldots < a_{N-1}$ and part of the payload p. First, we use division with remainder to split the payload into $qN! + r = p$ with a remainder $r \in \{0, \ldots, N! - 1\}$. While q is stored separately, we want to encode r as permutation σ. To this end, we establish a one-to-one mapping φ between the set of all permutations $\mathbb{S}_N$ and indices r:

$$\varphi : \mathbb{S}_N \to \{0, \ldots, N! - 1\}.$$

During encoding, our permutation coding determines the permutation $\sigma := \varphi^{-1}(r)$ and shuffles the tuple $a_0, \ldots, a_{N-1}$ using the inverse of this permutation, i.e. it stores

$$a_{\sigma^{-1}(0)}, \ldots, a_{\sigma^{-1}(N-1)}.$$

During decoding, this shuffled sequence gets sorted. Sorting effectively applies the permutation σ and in the process, we recover this permutation. Then the remainder of the payload r is computed using the index mapping φ, i.e. $r = \varphi(\sigma)$. In this manner, we have stored r without using any additional memory. We gain roughly $\log_2(N!)$ bits (see Table 1).

Algorithm 1 Generating a permutation.
Input: A Lehmer code r.
Output: The permutation $\sigma = \varphi^{-1}(r)$.

$\sigma(N-1) := 0$
For $i := 2, \ldots, N$:
  Perform division with remainder to get $ri + d := r$.
  $\sigma(N-i) := d$ // First entry set last to get a lexicographic order
  For $j := N+1-i, \ldots, N-1$:
    If $\sigma(j) \geq d$: $\sigma(j) := \sigma(j) + 1$ // Avoid collisions
Return σ.

Algorithm 2 Computing a Lehmer code.
Input: A permutation $\sigma \in \mathbb{S}_N$.
Output: The Lehmer code $r := \varphi(\sigma)$.

$r := 0$
$s := 2^N - 1$ // Bitmask of unseen indices
For $i := 0, \ldots, N-2$:
  $d :=$ Number of set bits in s from bit 0 to $\sigma(i) - 1$.
  $r := r + (N-1-i)!d$
  Unset bit $\sigma(i)$ of s.
Return r.

Table 1. Key quantities concerning the efficiency of our permutation coding. 13! does not fit into 32 bits.

Weight count $N+1$	2	3	4	5	6	7	8	9	10	11	12	13
$\log_2(N!)$	0	1	2.6	4.6	6.9	9.5	12.3	15.3	18.5	21.8	25.3	28.8
Sorting network size	0	1	3	5	9	12	16	19	25	29	35	39
Sorting network depth	0	1	3	3	5	5	6	6	7	9	8	9

To realize this scheme, we have to choose a mapping φ. That means we have to pick an ordering of the set of permutations $\mathbb{S}_N$. All $(N!)!$ possible choices would work but we seek one that lets us evaluate φ and φ^{-1} efficiently. We choose to order the permutations $\sigma \in \mathbb{S}_N$ through a lexicographic ordering of the index tuples $(\sigma(0), \ldots, \sigma(N-1))$. Algorithm 1 implements $\varphi^{-1}(r)$. It constructs the index tuple from right to left. Step i extracts a base-i digit d from the input r to choose the next entry. Since more significant digits determine entries further to the left, the ordering is indeed lexicographic. Algorithm 2 reverts this process by extracting digits from left to right. Its implementation with a bitmask is GPU friendly. These algorithms were described in more detail by Lehmer [1960] and first proposed by his father in 1906. Thus, we call $\varphi(\sigma)$ a Lehmer code.

The shuffling itself is a bit tricky to do on GPUs because when an array entry is accessed using a dynamically computed index, that causes costly register spilling. To generate the shuffled sequence $a_{\sigma^{-1}(0)}, \ldots, a_{\sigma^{-1}(N-1)}$, we write the indices $\sigma(0), \ldots, \sigma(N-1)$ into the most significant bits of $a_0, \ldots, a_{N-1}$ and then sort this sequence. To recover the permutation during sorting, we first write the indices $0, \ldots, N-1$ into the least significant bits of $a_{\sigma^{-1}(0)}, \ldots, a_{\sigma^{-1}(N-1)}$ (after shifting other bits to the left). After sorting, the least significant bits are $\sigma(0), \ldots, \sigma(N-1)$, i.e. the input to Algorithm 2. Sorting is done by optimal sorting networks [Knuth, 1998] (see Table 1).

3.4 Transforming the Blend Weights

Permutation coding shuffles the integers $a_0, \ldots, a_{N-1}$ arbitrarily. Therefore, they all have to be stored in the same manner and should cover roughly the same range. Vectors of weights $\mathbf{w} := (w_0, \ldots, w_{N-1})^\mathsf{T} \in \mathbb{R}^N$ do not meet this requirement. Equation (1) defines $N+1$ inequalities. Each of these inequalities defines a half space and together they characterize an N-dimensional simplex where each facet corresponds to one inequality. We seek a linear transformation that makes this simplex cover a large portion of the unit hypercube $[0,1]^N$ but keeps the weights sorted.

First, we characterize this simplex by determining its $N+1$ vertices. To this end, we seek the point where weight w_k for $k \in \{0, \ldots, N\}$ is maximal. That is accomplished by setting all smaller weights to zero while distributing the remaining weight equally. Then the vertex is

$$\mathbf{v}_k := \Big(\underbrace{0, \ldots, 0}_{k \text{ times}}, \frac{1}{N+1-k}, \ldots, \frac{1}{N+1-k} \Big)^\mathsf{T} \in \mathbb{R}^N.$$

In the special case $N = 3$, the simplex is a tetrahedron [Kuth and Meyer, 2021] (Fig. 1 left).

Now consider the linear transform defined by

$$u_i := (N+1-i) w_i + \sum_{j=0}^{i-1} w_j, \tag{2}$$

where $i \in \{0, \ldots, N-1\}$. If $\mathbf{w} = \mathbf{v}_k$ and $i < k$, we find

$$u_i = (N+1-i)0 + \sum_{j=0}^{i-1} 0 = 0.$$

On the other hand, if $\mathbf{w} = \mathbf{v}_k$ and $i \geq k$, we get

$$u_i = \frac{N+1-i}{N+1-k} + \sum_{j=k}^{i-1} \frac{1}{N+1-k} = \frac{N+1-i}{N+1-k} + \frac{i-k}{N+1-k} = 1.$$

Thus, this linear transform maps the vertex $\mathbf{v}_k$ to

$$(\underbrace{0, \ldots, 0}_{k \text{ times}}, 1, \ldots, 1)^\mathsf{T} \in [0,1]^N.$$

These vectors are exactly the corners of the unit hypercube $[0,1]^N$ at which the coordinates happen to be sorted. Thus, the simplex of valid weight vectors gets mapped to the larger simplex of sorted tuples in the unit hypercube (Fig. 1 middle). After this transform, each coordinate u_i covers the full range $[0,1]$ as intended. Furthermore, by shuffling entries of the vector $\mathbf{u} := (u_0, \ldots, u_{N-1})^\mathsf{T}$, we can attain any point in the unit hypercube, which is an indication that this scheme could be optimal (Fig. 1 right).

Appendix B proves that the following formula provides the inverse of the above transform:

$$w_i = \frac{1}{N+1-i} u_i - \sum_{j=0}^{i-1} \frac{1}{(N+1-j)(N-j)} u_j. \tag{3}$$

Both transforms take only linear time to compute since the sums for different outputs share common prefixes.

3.5 Error Analysis

In the next section, we turn the continuous value u_i into an integer $a_i B_i + b_i$ through quantization. This integer is fed to permutation coding. However, we should first understand how rounding errors in $\mathbf{u}$ affect the blend weights $w_0, \ldots, w_N$. As error metric, we choose the 2-norm. Crucially, we account for all weights in the error. Although w_N is not stored explicitly, errors in w_N harm the quality just as much as errors in any other weight. Prior work only accounts for errors in $w_0, \ldots, w_{N-1}$ [Kuth and Meyer, 2021].

Let $\tilde{\mathbf{w}} := (\tilde{w}_0, \ldots, \tilde{w}_{N-1})^\mathsf{T} \in \mathbb{R}^N$ denote the rounding errors in $\mathbf{w}$. If we enforce $w_N = 1 - \sum_{i=0}^{N-1} w_i$, we get $\tilde{w}_N := -\sum_{i=0}^{N-1} \tilde{w}_i$ as error for the greatest weight. Then the squared 2-norm of the error is

$$\sum_{i=0}^{N} \tilde{w}_i^2 = \|\tilde{\mathbf{w}}\|_2^2 + \tilde{w}_N^2 = \|\tilde{\mathbf{w}}\|_2^2 + \left(\sum_{i=0}^{N-1} \tilde{w}_i \right)^2 . \tag{4}$$

It accounts for the error in w_N explicitly. In Appendix C, we prove that this norm can be expressed in a surprisingly convenient manner in terms of the rounding errors in $\mathbf{u}$, $\tilde{\mathbf{u}} := (\tilde{u}_0, \ldots, \tilde{u}_{N-1})^\mathsf{T} \in \mathbb{R}^N$:

$$\|\tilde{\mathbf{w}}\|_2^2 + \left(\sum_{i=0}^{N-1} \tilde{w}_i \right)^2 = \sum_{i=0}^{N-1} \left(\frac{\tilde{u}_i}{\sqrt{(N+1-i)(N-i)}} \right)^2 . \tag{5}$$

We only have to scale each individual entry of $\tilde{\mathbf{u}}$ and then the norm that accounts for w_N happens to be the 2-norm. That means that it is optimal to quantize each entry of $\mathbf{u}$ individually but the precision should depend on the index $i \in \{0, \ldots, N-1\}$.

This result is intriguing because two problems cancel each other. Introducing a rounding error to each entry of $\mathbf{w}$ separately is suboptimal because it neglects the impact on w_N. Besides, the vectors $\mathbf{w}$ cannot fill the unit hypercube $[0, 1]^N$, even after shuffling. However, transforming $\mathbf{w}$ into $\mathbf{u}$ to address the latter problem simultaneously allows us to quantize each entry of $\mathbf{u}$ separately.

3.6 Quantization

With this error analysis, we are prepared to quantize $u_0, \ldots, u_{N-1}$ in a nearly optimal fashion. Two aspects require special care. Firstly, entries of $\mathbf{u}$ can be equal but the quantized values a_i entering permutation coding must be strictly ordered, i.e. $a_0 < \ldots < a_{N-1}$. Besides, the integers $a_0, \ldots, a_{N-1}$ should all cover the same range but we need different precision for different entries of $\mathbf{u}$.

To allow for different precision, we define precision factors $B_0, \ldots, B_{N-1} \in \mathbb{N}$ per entry. Then u_i is stored by $a_i \in \{0, \ldots, A-1\}$, where $A \in \mathbb{N}$ with $A > N$, and a less significant extra value $b_i \in \{0, \ldots, B_i - 1\}$. We determine a_i and b_i through division with remainder such that

$$a_i B_i + b_i = \left\lfloor (A - N) B_i u_i + (i + 1) B_i - \frac{1}{2} \right\rfloor . \tag{6}$$

This formula is carefully designed to stay within the allowable range:

$$(A - N) B_i u_i + (i + 1) B_i - \frac{1}{2} \geq (i + 1) B_i - \frac{1}{2} \geq 0,$$
$$(A - N) B_i u_i + (i + 1) B_i - \frac{1}{2} \leq (A + i + 1 - N) B_i - \frac{1}{2} < A B_i.$$

As long as $B_0 \leq \ldots \leq B_{N-1}$, we also get $a_{i+1} > a_i$ for all $i \in \{0, \ldots, N-2\}$ because

$$(A - N) B_{i+1} u_{i+1} + (i + 1 + 1) B_{i+1} - \frac{1}{2}$$
$$\geq (A - N) B_i u_i + (i + 1) B_i - \frac{1}{2} + B_i.$$

Algorithm 3 Encoder for blend attributes.
Input: A weight vector $\mathbf{w} \in \mathbb{R}^N$ satisfying Equation (1) and a tuple index $t \in \{0, \dots, T-1\}$.
Output: A code $c \in \mathbb{N}_0$.

Transform $\mathbf{w}$ into $\mathbf{u}$ according to Equation (2).
Quantize each u_i into a_i, b_i according to Equation (6).
Encode $t, b_0, \dots, b_{N-1}$ into the payload $p \in \mathbb{N}$.
Perform division with remainder to get $qN! + r := p$.
Construct $\sigma := \varphi^{-1}(r)$ using Algorithm 1.
Shuffle $a_0, \dots, a_{N-1}$ to get $a_{\sigma^{-1}(0)}, \dots, a_{\sigma^{-1}(N-1)}$.
Encode $q, a_{\sigma^{-1}(0)}, \dots, a_{\sigma^{-1}(N-1)}$ into c and return c.

Thus, the sequence entering permutation coding is strictly ordered.

Dequantization for $i \in \{0, \dots, N-1\}$ is done through

$$u_i = \frac{a_i B_i + b_i + 1 - (i+1)B_i}{(A-N)B_i} \tag{7}$$

This solution implements rounding to the nearest value. The boundary values zero and one are represented without error.

The precision factors should be roughly antiproportional to the error scaling factors from Equation (5). We pick them through a brute-force search, which minimizes the worst-case error defined by Equation (9). In each search, we prescribe the total number of bits and the minimal number of supported tuple indices. With this optimizer, we have prepared 291 reasonable parameter sets for our codec covering $N \in \{1, \dots, 12\}$ (see the supplemental C code). E.g. for eight weights with 5040 tuple indices stored in 48 bits, we use $A = 64$ and

$$(B_0, \dots, B_7) = (1, 1, 1, 2, 2, 3, 5).$$

3.7 Encoder and Decoder

With these considerations, our method is nearly complete. Algorithms 3 and 4 assemble the pieces into a complete codec. The encoder has two steps where it encodes integers $c_0, \dots, c_{K-1}$ with $c_i \in \{0, \dots, C_i - 1\}$ for all $i \in \{0, \dots, K-1\}$ into a single integer. That is done in linear time through repeated multiplication and addition:

$$c := \sum_{i=0}^{K-1} c_i \prod_{j=i+1}^{K-1} C_j = (((c_0 C_1 + c_1)C_2 + c_2) \cdots)C_{K-1} + c_{K-1}$$

The decoder has to undo these steps. To this end, we perform repeated division with remainder in reverse order. Note that the C_i are compile time constants. If some of them are chosen as powers of two, the compiler will implement the division through a right shift. We reward that in our brute force search by allowing 0.7% greater error for each such division. This choice leads to considerably more power-of-two divisors at the expense of tiny increases in error. For other divisors, compilers perform similar optimizations but they are more costly nonetheless.

Our GLSL implementation supports encoding into up to 64 bits, i.e. into two 32-bit unsigned integers. Multiplication and addition with carry are natively supported by GLSL through `umulExtended()` and `uaddCarry()`. For division with remainder by a number $C_i < 2^{16}$, we implement a multiword division that operates on 16 bits at a time and implements carry using the less significant 16 bits of a 32-bit integer [Warren, 2012]. Our codec only uses 64-bit operations for the steps where the most significant 32 bits are potentially non-zero.

Algorithm 4 Decoder for blend attributes.
Input: A code $c \in \mathbb{N}_0$.
Output: A weight vector $\mathbf{w} \in \mathbb{R}^N$ and a tuple index $t \in \{0, \dots, T-1\}$.

Decode $q, a_{\sigma^{-1}(0)}, \dots, a_{\sigma^{-1}(N-1)}$ from c.
Sort $a_{\sigma^{-1}(0)}, \dots, a_{\sigma^{-1}(N-1)}$ to recover $a_0, \dots, a_{N-1}$ and σ.
Compute $r := \varphi(\sigma)$ using Algorithm 2.
Decode $t, b_0, \dots, b_{N-1}$ from the payload $p := qN! + r$.
Dequantize each u_i from a_i, b_i according to Equation (7).
Transform $\mathbf{u}$ into $\mathbf{w}$ through Equation (3).
Return $\mathbf{w}, t$.

Table 2. Potential savings in bits per vertex when sorting bone indices instead of blend weights for the scene in Fig. 4 (201 k vertices) reduced to different numbers of weights per vertex. The benefit of a smaller table is not enough to outweigh the increased cost for storing weights. The last row evaluates our optimizations from Sec. 3.1.

Weight count $N+1$	4	6	8
Table size T (sorted indices)	2327	2433	2467
Table size T (sorted weights)	5977	6465	6482
Saving for the table	0.12	0.19	0.26
Saving for the tuple index	1.36	1.41	1.39
Saving for the weights	-4.58	-9.49	-15.30
T as in Kuth and Meyer [2021]	9539	11187	11329

4 RESULTS

We now evaluate our technique. We start with data justifying our choice to sort blend weights instead of bone indices (Sec. 4.1). Then we analyze the worst-case error of our technique in comparison to prior work [Kuth and Meyer, 2021] with regard to the norm in Equation (4). Errors in vertex positions of actual models reflect these theoretical numbers (Sec. 4.2). Finally, we report frame times (Sec. 4.3) and timings for encoding and decoding on its own (Sec. 4.4).

4.1 Sorting Bone Indices

Recall that we can sort pairs of weights and bone indices either by weight or by index. We chose sorting by weight but evaluate the alternative here. Sorting by bone indices makes the table of bone indices smaller (Table 2 fourth row). Additionally, the smaller range of tuple indices requires less storage per vertex (Table 2 fifth row).

On the other hand, unsorted weights are only constrained by the inequalities $0 \leq w_0, \dots, w_{N-1}$ and $\sum_{i=0}^{N-1} w_i \leq 1$. This simplex has volume $\frac{1}{N!}$ compared to a volume of

$$\frac{1}{N!} \prod_{i=0}^{N-1} \frac{1}{N+1-i} = \frac{1}{N!(N+1)!}$$

for the simplex in Equation (1). Thus, we expect a theoretically optimal encoding for non-sorted weights, which does not exist yet, to take $\log_2((N+1)!)$ more bits than a theoretically optimal coding for sorted weights, which we get close to (Table 2 sixth row). This increased cost for the weights is considerably greater than the potential savings due to a smaller table. We also find that the optimizations in Sec. 3.1 nearly halve the table size (Table 2 last row).

Table 3. 2-norm errors across all weights (see Equations (8) and (9)) for different blend attribute compression techniques. All values were scaled by 1000 to improve readability. Our technique always has the lowest error and the advantage grows considerably for more weights.

Bit count	24	32	32	48	48	48	48
Weight count $N+1$	4	4	5	6	7	8	9
Supported table size T	1024	1024	2048	4096	2048	8192	4096
Unit cube sampling [2021]	115.47	13.64	72.13	21.56	51.43	120.70	282.84
POT AABB [2021]	27.94	6.82	36.07	10.78	25.72	60.35	68.43
Any AABB [2021]	23.73	3.72	17.75	5.00	10.87	25.63	37.55
OSS [2021]	13.53	2.06	-	-	-	-	-
Permutation coding, ours	9.28	1.34	4.97	1.00	1.78	3.70	4.85

Bit count	64	64	64	64	106
Weight count $N+1$	10	11	12	13	13
Supported table size T	8192	8192	8192	8192	8192
Unit cube sampling [2021]	153.01	169.16	382.97	416.33	49.17
POT AABB [2021]	37.65	41.62	92.65	100.73	6.11
Any AABB [2021]	17.38	26.49	37.30	48.79	4.42
Permutation coding, ours	1.82	2.45	3.20	4.40	-

4.2 Worst-Case Error

The methods proposed in prior work [Kuth and Meyer, 2021] all strive to get the same maximal error $\frac{1}{2}\Delta > 0$ for each entry of $\mathbf{w}$ independently. Thus, the error with respect to the norm given in Equation (4) is

$$\sqrt{\sum_{i=0}^{N-1}\left(\frac{1}{2}\Delta\right)^2 + \left(\sum_{i=0}^{N-1}\frac{1}{2}\Delta\right)^2} = \frac{\Delta}{2}\sqrt{N+N^2}. \tag{8}$$

If we were to disregard the error in w_N, the 2-norm error would be only $\frac{\Delta}{2}\sqrt{N}$. That emphasizes the importance of an error analysis minimizing the error across all weights (Sec. 3.5).

According to Equation (7), the maximal error in u_i is $\frac{1}{2(A-N)B_i}$. Now Equation (5) yields the worst-case error for our approach:

$$\frac{1}{2(A-N)}\sqrt{\sum_{i=0}^{N-1}\frac{1}{(N+1-i)(N-i)B_i^2}}. \tag{9}$$

Table 3 compares these worst-case errors for different techniques operating at different bit counts and with different table sizes T. Even the best prior work for $N=3$ [Kuth and Meyer, 2021] has a 46% to 53% greater error than our permutation coding. For greater N, this OSS is not applicable and the advantage of our work is still greater. It grows with $\log_2(N!)$. For 13 weights, our error is an order of magnitude smaller than that of the best prior work and two orders of magnitude smaller than that of unit cube sampling (i.e. simple fixed-point quantization of $w_0, \ldots, w_{N-1} \in [0,1]$). Halving the error takes roughly N bits, so it would take another 42 bits to reach the same error with any AABB [Kuth and Meyer, 2021].

Of course, the improved worst-case error compared to OSS [Kuth and Meyer, 2021] is due to our chosen error metric. With regard to the error in $w_0, \ldots, w_{N-1}$ alone, OSS is optimal by design. Fig. 3 demonstrates that our metric, which additionally accounts for the largest weight w_N, is

Fig. 3. Vertex positions for this skinned character (1.9 m tall) have been computed using compressed weights and weights provided as 32-bit floats. All compression techniques use four bytes for four weights and the tuple index. We color code the error in the world space positions. Model from mixamo.

indeed more relevant. The rounding errors in vertex positions are quite noisy but errors of our permutation coding tend to be lower than with OSS at equal bit count.

4.3 Frame Times

We measure frame times using Vulkan on a desktop with an Intel Core i5-9600K and an NVIDIA RTX 2070 Super with 8 GB VRAM. We have merged 28 character models into one scene with 1412 bones. Therefore, bone indices use 16 bits. This scene gets instanced 50 times for a total of 100 million vertices (Fig. 4). Each vertex stores fixed-point positions and texture coordinates and an octahedral normal [Meyer et al., 2010] in $64 + 32 + 32 = 128$ bits. All vertex data is bound through a single vertex buffer. The bone transforms are stored as fixed-point unit quaternions, translations and isotropic scalings in $8 \cdot 16 = 128$ bits each. An earlier version used 12 floats for a 3×4 matrix but then L1 cache bandwidth became limiting and the differences between different blend attribute compression techniques were smaller.

Table 4 shows the results. First of all, we note that 48 bits per vertex are more costly than 64 bits, presumably due to alignment issues. That would change if the other vertex attributes were 16 bits larger, e.g. due to tangent frames. Our ground truth stores all weights as 32-bit floats and indices as 16-bit integers per vertex. Thus, it is already the slowest method for four weights and the cost grows quickly.

At 32 bits with four weights, our permutation coding is slightly slower than unit cube sampling and POT AABB, but at a much lower error. It outperforms OSS in terms of error and speed. Unit cube sampling with 64 bits is slightly faster and much more accurate but the increased memory footprint is a drawback in itself. At higher bit rates, we make similar observations although the computational cost of our method becomes a bit more significant with increasing weight count. For 10 bones, our method is the only one that offers a reasonable error at 64 bits.

4.4 Compute Timings

Our frame times depend heavily on specifics of our renderer. Therefore, we also set up a benchmark focusing on the computational cost of OSS and our method (encoding and decoding). We run a compute shader and derive a 64-bit integer from its thread index. We test the decoder by feeding it this integer. There are no reads from buffers. To test the encoder, we turn the thread index into arbitrary weights and encode them. A few additional instructions tie the results to outputs so that the computation does not fall victim to dead-code elimination.

Fig. 4. Our benchmark scene with 1400 character models from mixamo, rendered at 1280×1024.

Table 4. Total frame times in milliseconds for rendering Fig. 4 with various techniques. Bit counts refer to weights and (tuple) indices, errors are as in Table 3.

	Bit count	$N+1$	Error	Time
Unit cube sampling [2021]	32	4	27.49	8.7
POT AABB [2021]		4	13.75	8.8
OSS [2021]		4	4.16	10.0
Permutation coding, ours		4	2.57	9.3
OSS [2021]	48	4	0.10	11.8
POT AABB [2021]		6	10.78	11.6
POT AABB [2021]		8	60.35	11.6
Permutation coding, ours		8	3.66	13.0
Unit cube sampling [2021]	64	4	0.01	9.0
POT AABB [2021]		8	7.34	10.7
POT AABB [2021]		10	37.65	10.9
Permutation coding, ours		8	0.67	12.6
Permutation coding, ours		10	1.76	14.6
Ground truth	192	4	0	10.6
Ground truth	288	6	0	14.3
Ground truth	384	8	0	15.9
Ground truth	480	10	0	20.4

Table 5 shows the results. We note that our technique for four weights, is considerably less expensive than OSS. The cost of our technique scales roughly linearly with the number of weights. Encoding is slightly faster than decoding. Although encoding is more commonly done on CPU, such a fast GPU implementation could come in handy for interactive editors or procedural content.

Table 5. Timings for the computations of encoding or decoding blend attributes in picoseconds per vertex. The combined overhead for the encoding and decoding tests (generating weights from the thread index and tying them to outputs) is reported separately.

$N+1$	4 (OSS)	4	4	5	6	7	8	9	10	11	12	13
Bit count	32	24	32	32	48	48	48	48	64	64	64	64
Encoder	-	10.2	10.3	13.7	22.3	26.2	32.7	37.3	49.2	56.5	65.2	73.6
Decoder	17.6	10.2	10.3	15.5	27.2	32.0	38.0	42.2	56.3	62.0	80.6	82.6
Overhead	5.4	5.4	5.4	6.4	7.2	8.5	9.6	10.5	11.7	12.6	13.8	14.9

Our table construction could also be implemented on GPU using standard parallel sorting and scanning procedures.

5 CONCLUSIONS

The quest for greater fidelity and more detail in real-time rendering is never ending. Nonetheless, it is still common to restrict artists to use only four bones per vertex. Our permutation coding makes it viable to lift this restriction. For example, supporting eight weights per vertex with table sizes up to $T = 8192$ and an accuracy equivalent to 10 bits per weight costs only 48 bits per vertex. This bandwidth is easily affordable and so is the computational cost.

Similar methods could be applied for compression of barycentric coordinates $\lambda_0, \dots, \lambda_N \geq 0$ with $\sum_{i=0}^{N} \lambda_i = 1$ (without sorting). The corresponding simplex has vertices at the canonical basis vectors $\mathbf{e}_0, \dots, \mathbf{e}_{N-1} \in \mathbb{R}^N$. Thus, the pendant for the transform in Equation (2) is $u_i := \sum_{j=0}^{i} \lambda_j$. The trick in Equation (5) does not carry over, so optimal quantization becomes more challenging but permutation coding would be as effective as for sorted blend weights.

REFERENCES

Niels Henrik Abel. 1826. Beweis der Unmöglichkeit, algebraische Gleichungen von höheren Graden als dem vierten allgemein aufzulösen. *Journal für die reine und angewandte Mathematik* 1, 1 (1826), 65–84. https://doi.org/10.1515/9783112347386-009

Marc Alexa. 2002. Linear Combination of Transformations. *ACM Trans. Graph.* 21, 3 (2002). https://doi.org/10.1145/566654.566592

Dean Calver. 2002. Vertex Decompression in a Shader. In *Direct3D ShaderX – Vertex and Pixel Shader Tips and Tricks*, Wolfgang F. Engel (Ed.). Wordware Publishing, Inc., 172–187.

Dean Calver. 2004. Using Vertex Shaders for Geometry Compression. In *ShaderX²: Shader Programming Tips and Tricks with DirectX 9.0*, Wolfgang F. Engel (Ed.). Wordware Publishing, Inc., 3–12.

Nicolas Fréchette. 2017. *Simple and Powerful Animation Compression.* https://www.gdcvault.com/play/1024009/Simple-and-Powerful-Animation Game Developers Conference.

Ivo Zoltan Frey and Ivo Herzeg. 2011. Spherical Skinning with Dual Quaternions and QTangents. In *ACM SIGGRAPH 2011 Talks.* https://doi.org/10.1145/2037826.2037841

Andrew Garrard. 2020. *Khronos Data Format Specification v1.3.1.* https://www.khronos.org/registry/DataFormat/specs/1.3/dataformat.1.3.html#_compressed_texture_image_formats

Jean Geffroy, Axel Gneiting, and Yixin Wang. 2020. Rendering the Hellscape of Doom Eternal. In *ACM SIGGRAPH '20: ACM SIGGRAPH 2020 Courses.* https://advances.realtimerendering.com/s2020

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. Nanite — A Deep Dive. In *ACM SIGGRAPH '21: ACM SIGGRAPH 2021 Courses.* http://advances.realtimerendering.com/s2021

Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. 2008. Geometric Skinning with Approximate Dual Quaternion Blending. *ACM Trans. Graph.* 27, 4 (2008). https://doi.org/10.1145/1409625.1409627

Benjamin Keinert, Matthias Innmann, Michael Sänger, and Marc Stamminger. 2015. Spherical Fibonacci Mapping. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 34, 6 (2015). https://doi.org/10.1145/2816795.2818131

Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3 - Sorting and Searching, 2nd Edition.* Addison-Wesley Professional.

Bastian Kuth and Quirin Meyer. 2021. Vertex-Blend Attribute Compression. In *High-Performance Graphics - Symposium Papers.* The Eurographics Association. https://doi.org/10.2312/hpg.20211282 Best paper.

Binh Huy Le and Zhigang Deng. 2013. Two-Layer Sparse Compression of Dense-Weight Blend Skinning. *ACM Trans. Graph. (Proc. SIGGRAPH)* 32, 4 (2013). https://doi.org/10.1145/2461912.2461949

Binh Huy Le and Jessica K. Hodgins. 2016. Real-Time Skeletal Skinning with Optimized Centers of Rotation. *ACM Trans. Graph. (Proc. SIGGRAPH)* 35, 4 (2016). https://doi.org/10.1145/2897824.2925959

Binh Huy Le and J P Lewis. 2019. Direct Delta Mush Skinning and Variants. *ACM Trans. Graph. (Proc. SIGGRAPH)* 38, 4 (2019). https://doi.org/10.1145/3306346.3322982

Binh Huy Le, Keven Villeneuve, and Carlos Gonzalez-Ochoa. 2021. Direct Delta Mush Skinning Compression with Continuous Examples. *ACM Trans. Graph. (Proc. SIGGRAPH)* 40, 4 (2021). https://doi.org/10.1145/3450626.3459779

Derrick H. Lehmer. 1960. Teaching combinatorial tricks to a computer. *Proceedings of Symposia in Applied Mathematics* 10 (1960), 179–193. https://doi.org/10.1090/psapm/010

Adrien Maglo, Guillaume Lavoué, Florent Dupont, and Céline Hudelot. 2015. 3D Mesh Compression: Survey, Comparisons, and Emerging Trends. *ACM Comput. Surv.* 47, 3 (2015). https://doi.org/10.1145/2693443

Nadia Magnenat-Thalmann, Richard Laperrière, and Daniel Thalmann. 1989. Joint-Dependent Local Deformations for Hand Animation and Object Grasping. In *Proceedings on Graphics Interface '88*. Canadian Information Processing Society.

David K. McAllister, Alexandre Joly, and Peter Tong. 2014. Lossless Frame Buffer Color Compression. United States Patent 8670613.

Quirin Meyer, Jochen Süßmuth, Gerd Sußner, Marc Stamminger, and Günther Greiner. 2010. On Floating-Point Normal Vectors. *Computer Graphics Forum (Proc. EGSR)* 29, 4 (2010). https://doi.org/10.1111/j.1467-8659.2010.01737.x

Jorn Nystad, Anders Lassen, Andy Pomianowski, Sean Ellis, and Tom Olson. 2012. Adaptive Scalable Texture Compression. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association. https://doi.org/10.2312/EGGH/HPG12/105-114

Emil Persson. 2012. Creating Vast Game Worlds: Experiences from Avalanche Studios. In *ACM SIGGRAPH 2012 Talks*. Article 32. https://doi.org/10.1145/2343045.2343089

Budirijanto Purnomo, Jonathan Bilodeau, Jonathan D. Cohen, and Subodh Kumar. 2005. Hardware-Compatible Vertex Compression Using Quantization and Simplification. In *Graphics Hardware*. The Eurographics Association. https://doi.org/10.2312/EGGH/EGGH05/053-062

Sylvain Rousseau and Tamy Boubekeur. 2020. Unorganized Unit Vectors Sets Quantization. *Journal of Computer Graphics Techniques (JCGT)* 9, 4 (2020). https://jcgt.org/published/0009/04/02/

Henry S. Jr. Warren. 2012. *Hacker's Delight, 2nd Edition.* Addison-Wesley Professional.

A LEMMA FOR THE FOLLOWING PROOFS

The proofs in the appendix rely on the following equation:

$$\sum_{k=j+1}^{i-1} \frac{1}{(N+1-k)(N-k)} = \frac{1}{N+1-i} - \frac{1}{N-j} \tag{10}$$

Proof. We proceed by induction over $i-j$. For $i-j=1$,

$$\sum_{k=j+1}^{i-1} \frac{1}{(N+1-k)(N-k)} = 0 = \frac{1}{N+1-i} - \frac{1}{N-j}.$$

Then if Equation (10) holds for sums with one term less, we find:

$$\begin{aligned}
&\sum_{k=j+1}^{i-1} \frac{1}{(N+1-k)(N-k)} \\
=&\frac{1}{(N+2-i)(N+1-i)} + \sum_{k=j+1}^{i-2} \frac{1}{(N+1-k)(N-k)} \\
\overset{(10)}{=}&\frac{1}{(N+2-i)(N+1-i)} + \frac{1}{N+2-i} - \frac{1}{N-j} \\
=&\frac{1}{N+1-i} - \frac{1}{N-j}
\end{aligned}$$

□

B INVERSE TRANSFORM

Proof. To prove Equation (3), we substitute Equation (2) into its right-hand side and apply Equation (10):

$$\begin{aligned}
&\frac{1}{N+1-i}u_i - \sum_{j=0}^{i-1}\frac{1}{(N+1-j)(N-j)}u_j \\
=&w_i + \frac{1}{N+1-i}\sum_{j=0}^{i-1}w_j - \sum_{j=0}^{i-1}\frac{1}{N-j}w_j - \sum_{\substack{j,k=0\\ j<k}}^{i-1}\frac{1}{N+1-k}\frac{1}{N-k}w_j \\
=&w_i + \sum_{j=0}^{i-1}\left(\frac{1}{N+1-i} - \frac{1}{N-j} - \sum_{k=j+1}^{i-1}\frac{1}{(N+1-k)(N-k)}\right)w_j \\
\overset{(10)}{=}&w_i
\end{aligned}$$

□

C ERROR METRIC

Proof. To prove Equation (5), we start at the right-hand side, substitute in Equation (2), expand and apply Equation (10):

$$\begin{aligned}
&\sum_{i=0}^{N-1}\frac{1}{(N+1-i)(N-i)}\tilde{u}_i^2 \\
=&\sum_{i=0}^{N-1}\frac{1}{(N+1-i)(N-i)}\left((N+1-i)\tilde{w}_i + \sum_{j=0}^{i-1}\tilde{w}_j\right)^2 \\
=&\sum_{i=0}^{N-1}\frac{N+1-i}{N-i}\tilde{w}_i^2 + 2\sum_{i=0}^{N-1}\sum_{j=0}^{i-1}\frac{1}{N-i}\tilde{w}_i\tilde{w}_j + \sum_{i=0}^{N-1}\sum_{j,k=0}^{i-1}\frac{1}{(N+1-i)(N-i)}\tilde{w}_j\tilde{w}_k \\
=&\sum_{i=0}^{N-1}\left(\frac{N+1-i}{N-i} - \frac{1}{N-i}\right)\tilde{w}_i^2 + \sum_{i,j=0}^{N-1}\frac{\tilde{w}_i\tilde{w}_j}{N-\max\{i,j\}} + \sum_{j,k=0}^{N-1}\sum_{i=\max\{j,k\}+1}^{N-1}\frac{1}{(N+1-i)(N-i)}\tilde{w}_j\tilde{w}_k \\
\overset{(10)}{=}&\sum_{i=0}^{N-1}\tilde{w}_i^2 + \sum_{i,j=0}^{N-1}\frac{\tilde{w}_i\tilde{w}_j}{N-\max\{i,j\}} + \sum_{j,k=0}^{N-1}\left(\frac{1}{N+1-N} - \frac{1}{N-\max\{j,k\}}\right)\tilde{w}_j\tilde{w}_k \\
=&\sum_{i=0}^{N-1}\tilde{w}_i^2 + \sum_{j,k=0}^{N-1}\tilde{w}_j\tilde{w}_k
\end{aligned}$$

□