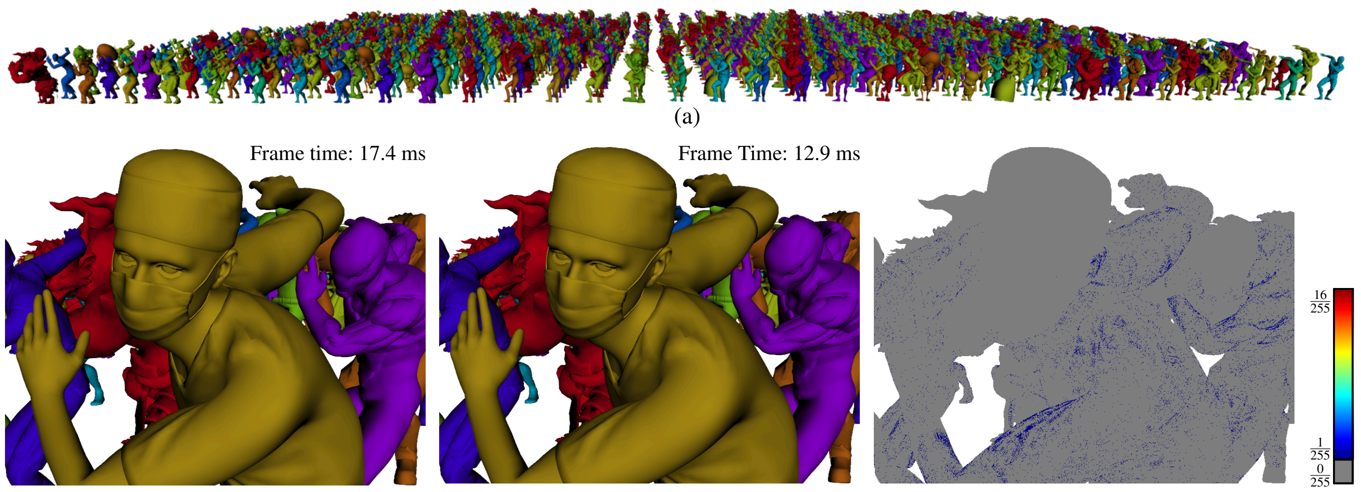
High-Performance Graphics (2021)
N. Binder and T. Ritschel (Editors)

Vertex-Blend Attribute Compression

Bastian Kuth and Quirin Meyer
Coburg University of Applied Sciences and Arts, Germany



(a)

(b) Uncompressed, 160 bpv

(c) Our Compression, 33 bpv

(d) Difference Image

Figure 1: *(a) Each vertex of the 1024 characters is animated by four weighted transformations. (b) Uncompressed, weights consume* 128 *bits per vertex (bpv) and bone indices* 32 *bpv. (c) In this lossy example, we compress weights to* 22 *bpv, bone indices to* 10 *bpv, require* 1 *bpv on average for a lookup table, and render faster. (d) The difference image shows that our compression produces only little errors. The color-scale on the right encodes the L^∞-norm of the pixel differences using normalized color intensities.*

Abstract

Skeleton-based animations require per-vertex attributes called vertex-blend attributes. They consist of a weight tuple *and a* bone index tuple. *With meshes becoming more complex, vertex-blend attributes call for compression. However, no technique exists that exploits their special properties. To this end, we propose a novel and optimal weight compression method called* Optimal Simplex Sampling *and a novel bone index compression. For our test models, we compress bone index tuples between 2.3:1 and 3.5:1 and weight tuples between 1.6:1 and 2.5:1 while being visually lossless. We show that our representations can speed rendering and reduces GPU memory requirements over uncompressed representations with a similar error. Further, our representations compress well with general-purpose codecs making them suitable for offline-storage and streaming.*

CCS Concepts

• ***Computing methodologies*** → *Rendering;* ***Animation;*** • ***Theory of computation*** → ***Data compression;***

1. Introduction

Advances in computer graphics continue to stress hardware boundaries. Screen resolutions get bigger, textures grow in size and number, and geometry becomes unprecedentedly complex with sensors and algorithms delivering more and more data. This development results in more triangles, vertices, and per-vertex attributes. While central processing units (CPUs) and graphics processing units (GPUs) reach higher compute performance levels with every new generation, memory performance grows much slower. To counteract this divergent performance growth of compute and bandwidth, compression becomes key. Modern GPUs support compressed textures and many internal compression techniques. By that, they save memory and increase the effective bandwidth.

Attribute compression is commonly found in practice to meet memory and performance requirements [FH11, Per12, GGW20]. However, it mostly does not go beyond naïve uniform quantiza-

DOI: 10.2312/hpg.20211282

tion techniques, missing out a lot of compression potential. Better compression is achieved with techniques designed for specific attributes, but, to our knowledge, there is no method that handles vertex-blend attributes well. Vertex-blend attributes are memory intense and their compression is worthwhile: They consist of an n-tuple of bone indices an n-tuple of weights, where $n = 4$ in practice. Uncompressed, they make ca. 38% the per-vertex memory of Fig. 1. However, compression must not degrade image quality. Weights are carefully computed or determined by an artist. Unwary quantization may result in unpleasant artifacts. Additionally, bone indices must be compressed without any loss.

To our knowledge, no documented vertex-blend attribute compression method exists. We compress independent and identically distributed (i.i.d.) vertex-blend attributes and **contribute**

1. a novel, lossless bone index compression,
2. an analysis of the vertex-blend weight space,
3. a broad overview of weight compression methods, and
4. Optimal Simplex Sampling (OSS), a novel and optimal compression scheme for weights.

For each vertex, we guarantee a time complexity of $\mathcal{O}(1)$. Decompression is fast and can speed rendering times. Our weight compression can be lossless over fixed-point weights and visually lossless over floating-point weights. A controllable error lets us trade quality against compression ratio to achieve lossy compression (cf. Fig. 1). Compression is fast enough to have little performance impact on asset preparation. We do not affect triangle or vertex order making it compatible with many mesh pre-processing techniques. OSS compresses well with general purpose codecs enabling offline-storage and streaming. Our compression works with vertex-blend methods that assume convex weights, such as linear blend skinning [MTLT89] or dual quaternion skinning [KCvO08]. While most of our compression methods handle an arbitrary number of weights, OSS supports only up to four weights, which is sufficient for most practical situations.

2. Related Work

Compression and decompression are frequently used in computer graphics. Due to the topic's broadness, we focus on previous work related to real-time GPU rasterization methods in Sec. 2.1 and give a brief overview of skeleton-based animations in Sec. 2.2.

2.1. GPU-Related Compression Methods

A significant amount of GPU memory consumption is due to textures. Hence, most GPUs support lossy *texture compression* with random read-access. It provides memory space reduction, better bandwidth utilization, faster sampling, and lower power dissipation [NLP*12]. Most formats [Gar19] subdivide textures into equal-sized blocks. Compression ratios range from 2:1 to 36:1 [NLP*12]. *Super-compressed* textures methods [KPM16, Bin21] compress already compressed textures. This reduces disk-space and speeds CPU-GPU data transfer. A GPU compute-shader decodes super-compressed to conventionally compressed textures. Similar approaches decode on the CPU [Bin20] or decode to uncompressed textures on the GPU [OBGB11].

Several special *GPU buffer compression techniques* exist for depth [HAM06] and float buffers [SWR*08, PLS12, NVB*20]. Seiler et. al propose to unify CPU and GPU virtual memory in the presence of compressed and swizzled GPU buffers [SLY20]. Recently, Sakharnykh and colleagues [SLK20] implemented a GPU LZ4 codec [Col19] to speed memory transfers between multiple GPUs. GPUs have several buffer compression techniques built-in [Bre16, MJT14]. Those methods are mostly lossless or visually lossless and some are transparent to the developer. The main purpose is not to reduce memory consumption, but to increase bandwidth efficiency and reduce power dissipation.

Vertex-blend attribute compression is part of *geometry compression*, pioneered by Deering [Dee95]. Multiple reports [PKJ05, AG05, MLDH15] provide overviews over the vast field. Meyer et al. [MKSS12] compress *triangle topology* to about 3.7–7.6 bits per triangle (bpt). Their GPU approach decompresses index buffers every frame without significant performance loss. Jakob et al. [JBG17] provide a data-parallel codec that compresses triangles to 3.5–4.2 bpt. It runs fast enough on the GPU to reduce CPU-GPU data transfer times. Kubisch [Kub20] uses mesh shaders [Mic20] to decompress topology. He subdivides a mesh into *meshlets* of 64 vertices with a meshlet-local index buffer of 84 triangles and achieves a triangle compression ratio of 3:1.

Vectors of GPU vertex buffers have up to four components and can be quantized at fixed bit-levels [Khr21], only. Purnomo et al. [PBCK05] achieve more flexibility by quantizing attributes with arbitrary many bits. However, they do not exploit the individual compressible properties for each attribute. Kwan et al. [KXW*18] store vertex attributes in a compressed texture. They permute the vertex order such that the compression error gets sufficiently small. Meyer et al. [MSGS11] use a view-dependent approach to adjust the number of bits for positions by adding or removing bit-planes. They yield compression ratios from 2.5:1 to 4:1 and render faster. Lee et al. [LCL10] partition a mesh into multiple sub-meshes, quantized with 8 bits. They prevent cracks by aligning the sub-meshes along a common grid. Jakob et al. [JBG17] compress positions between 2.7:1 and 3:1 with a data-parallel arithmetic codec. Special schemes for unit-vectors [MSS*10, CDE*14, KISS15, RB20] reach about 2:1 compression ratio without visual error and can decrease render times.

2.2. Skeleton-Based Animation

Vertex-blending (*skinning*, *skeleton-based*) *animation* transforms a mesh to perform gestures. An artist or automated process prepares a *skeleton* (*armature*, *rig*) for the mesh and assigns *vertex-blend* (*skinning*) attributes to its vertices. The skeleton is a tree hierarchy with M nodes called *bones*. For each bone $b \in \{1,\ldots,M\}$, we assign a *bone transformation* $\mathbf{A}_b \in \mathbb{T}$, such as rotation, translation, or scaling. Vertex-blend attributes consist of *weights* $\vec{w} = [w_1,\ldots,w_n]^T \in \mathbb{R}^n$ and *bone indices* $\vec{I} = [I_1,\ldots,I_n]^T \in \mathbb{N}^n$. To animate a mesh vertex from its *rest pose* $\vec{p} \in \mathbb{R}^{2,3,4}$ to its *deformed pose* $\vec{p}\,'$, we compute:

$$\vec{p}\,' = \mathbf{A} \otimes \vec{p} \quad \text{with} \quad \mathbf{A} = \sigma\big((w_1 \odot \mathbf{A}_{I_1}) \oplus \cdots \oplus (w_n \odot \mathbf{A}_{I_n})\big). \quad (1)$$

Thereby, $\mathbb{R} \odot \mathbb{T} \mapsto \mathbb{T}$ is a scalar multiplication, $\mathbb{T} \oplus \mathbb{T} \mapsto \mathbb{T}$ adds two transformations, and σ is a normalization mapping $\mathbb{T} \mapsto \mathbb{T}$.

Depending on the underlying method, different choices for $\mathbb{T}$ exist: For affine matrices [MTLT89], $\mathbb{T}$ is $\mathbb{R}^{3\times4}$, $\odot$ is a matrix scaling, $\oplus$ is a matrix addition, σ orthonormalizes the rotational part of the matrix, and $\otimes$ is a matrix-vector product. In case $\mathbb{T}$ is the set of dual quaternions [KCvO08], $\odot$ scales a dual quaternion, $\oplus$ adds dual quaternions, σ projects a dual to a unit dual quaternion, and $\otimes$ is the Hamiltonian product with the Euclidean point $\vec{p}$ represented as a dual quaternion. Common to those and other methods [Ale02, Kv05] is that the weights $\vec{w}$ are *convex*: they build a *partition of unit* $\sum_{i=1}^{n} w_i = 1$ and are *non-negative* $w_i \geq 0$.

Skeleton-based animation is efficient for GPU real-time rendering: First, an animation only adjusts the bone transformations $\mathbf{A}_b$. Weights and indices remain constant. Since the number of bones is small, changing transformations is fast. Second, the GPU programming models suits Eq. (1) and enables efficient implementations. Third, animations appear plausible already at a low number of vertex-blend attributes n, which keeps computation at moderate levels. For our test meshes of Tab. 1, 51% vertices require one, 22% two, 13% three, and 14% four weights. Even though game engines have recently started to support more weights, they recommend using four weights for performance reasons [Cry19, Uni21].

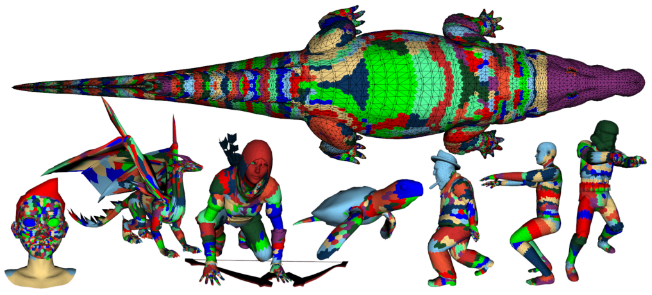
While vertex-blend attributes consume a significant amount of vertex data, it has not yet been explored how to efficiently compress them. Existing animation compression methods [SSK05, LDJ*19, Fré17] deal with bone transformations only.

3. Bone Index Compression

A vertex has a bone index tuple $\vec{I} \in \mathbb{N}^n$. Each $\vec{I}_i$ points at a bone transformation. For the test meshes of Tab. 1, we see that a mesh with M bones requires $n\lceil\log_2 M\rceil$ bits per bone. With four bones per vertex, we stay below 32 bpv (cf. col. *Raw*).

We improve compression by utilizing coherence in the bone index tuples. We observe that only a small number of combinations of bone indices are used. We call those combinations *unique tuples*. Tab. 1 assigns the same color to vertices of unique tuples. We place unique tuples in a lookup table (LUT). Instead of n bone indices, a vertex uses a single *tuple index* into that LUT. Column *Unique Tuples* of Tab. 1 shows that the number of unique tuples is surprisingly small. Each unique tuple is therefore addressable with the number of bits shown in column *Tuple Idx*. Note that we need to store the LUT, too. Splitting the LUT size over the vertices adds an additional per-vertex cost shown in column *LUT*. We store bone indices in the LUT with byte granularity. That and assuming 32 bpv for uncompressed bone indices, we compress bone indices at 2.3:1 to 3.5:1. Decompression time complexity for each bone index is $\mathcal{O}(1)$. Using a hash-map, compression has an average time complexity of $\mathcal{O}(1)$ per vertex.

As the order of vertex-blend attributes within a vertex is arbitrary, we could reduce the number of unique bone index tuples further by sorting the bone indices inside a tuple. Weights must be swapped accordingly. As weights require more memory, we receive better compression when sorting weights instead of bone indices (cf. Sec. 4). In fact, Tab. 1 already uses weight sorting.



Croc: Nile Crocodile Swimming by Monster, CC BY-NC-ND 4.0, https://skfb.ly/6Aotx, Face: Facial Rig test. by bayuitra, CC BY 4.0, https://skfb.ly/6tKXo, Dragon: Black Dragon with Idle Animation by 3DHaupt, CC BY-NC 4.0, https://skfb.ly/FWLt, Turtle: Model 50A - Hatchling Hawksbill sea turtle by DigitalLife3D, CC BY-NC 4.0, https://skfb.ly/6QTKp, Trooper: DANCING STORMTROOPER by StrykerDoesAnimation, CC BY 4.0, https://skfb.ly/WVIA

Mesh	V	M	Raw [bpv]	Unique Tuples	Tuple Idx [bpv]	LUT [bpv]
Croc	12800	108	28	1096	11	2.7
Face	11371	109	28	1093	11	3.0
Dragon	22844	121	28	1330	11	1.8
Archer	12424	54	24	354	9	0.9
Turtle	4346	24	20	148	8	1.1
Boss	5828	56	24	379	9	2.1
Human	20340	53	24	420	9	0.7
Trooper	5174	53	24	302	9	1.9

Table 1: *Bone Index Compression. Vertices within a mesh that have the same color use the same unique index tuple. For the meshes (col.* Mesh*), we list the vertex (col. V) and bone count (col. M). Col.* Raw *shows the bits per vertex (bpv) required for four uncompressed bone indices. We store combinations of bone indices in a LUT with* Unique Tuples *entries. Each LUT entry is addressable with* Tuple Idx *bits. Col.* LUT *shows the per-vertex overhead of the LUT.*

4. Weight Compression

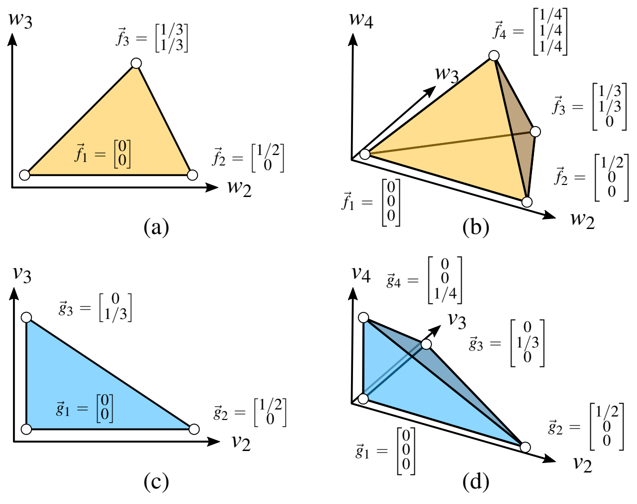
We review common weight representations (Sec. 4.1) and analyze the weight space (Sec. 4.2) in order to derive compact weight representations (Secs. 4.3–4.6). For readability, we abbreviate the methods as shown in parenthesis of the respective caption.

4.1. Naïve Quantization Methods

n Floats (Floats): Mostly, n floats are used for weights. This constitutes the uncompressed form and serves as baseline.

n Fixed-Point Numbers (Fixed): The dynamic range of floats is not required for bounded weights. Moreover, the space between two adjacent floats gets denser towards zero. We found no reason to sample small weights more densely than large ones. Thus, we sample $[0;1)$ with a $(B_\mathrm{m}+1)$-bit fixed-point number, where B_m is the number of mantissa bits of the float type. Since this conversion preserves the maximum quantization error of floats, it is quasi-lossless. To save the extra bit that only encodes the 1, we sample $[0;1]$ and accept a slightly larger error.

Unit Cube Sampling (Cube): Since weights are convex, we compute one weight from the others. The remaining weights form an $(n-1)$-D unit cube. Thus, we call this method *unit cube sampling*.



w_3
$\vec{f}_3 = \begin{bmatrix}1/3\\1/3\end{bmatrix}$
$\vec{f}_1 = \begin{bmatrix}0\\0\end{bmatrix}$
$\vec{f}_2 = \begin{bmatrix}1/2\\0\end{bmatrix}$
w_2
(a)
w_4
$\vec{f}_4 = \begin{bmatrix}1/4\\1/4\\1/4\end{bmatrix}$
w_3
$\vec{f}_3 = \begin{bmatrix}1/3\\1/3\\0\end{bmatrix}$
$\vec{f}_2 = \begin{bmatrix}1/2\\0\\0\end{bmatrix}$
$\vec{f}_1 = \begin{bmatrix}0\\0\\0\end{bmatrix}$
w_2
(b)
v_3
$\vec{g}_3 = \begin{bmatrix}0\\1/3\end{bmatrix}$
$\vec{g}_1 = \begin{bmatrix}0\\0\end{bmatrix}$
$\vec{g}_2 = \begin{bmatrix}1/2\\0\end{bmatrix}$
v_2
(c)
v_4
$\vec{g}_4 = \begin{bmatrix}0\\0\\1/4\end{bmatrix}$
v_3
$\vec{g}_3 = \begin{bmatrix}0\\1/3\\0\end{bmatrix}$
$\vec{g}_2 = \begin{bmatrix}1/2\\0\\0\end{bmatrix}$
$\vec{g}_1 = \begin{bmatrix}0\\0\\0\end{bmatrix}$
v_2
(d)

Figure 2: *Weight Spaces. We compute the largest weight w_1 from the smaller ones. (a) For three sorted convex weights, we store $w_2 \geq w_3$. Valid tuples $[w_2; w_3]^T$ form a triangle. (b) For four sorted convex weights, we keep $w_2 \geq w_3 \geq w_4$. Valid tuples form a tetrahedron. Delta weights shear the (c) triangle and (d) tetrahedron edges emanating from the origin onto the coordinate systems' axes.*

4.2. Weight Space Analysis

Vertex-blend weights are non-negative, sum to one, and their order is arbitrary. Thus, we arrange them in descending order with $w_i \geq w_{i+1}$ and compute $w_1 = 1 - \sum_{i=2}^{n} w_i$. We pick the largest weight w_1 as it consumes most bits. The remaining sorted weights form a triangle for three (cf. Fig. 2a), a tetrahedron for four (cf. Fig. 2b), and an $(n-1)$-D simplex for n weights with the inequalities:

$$w_2 + \sum_{i=2}^{n} w_i \leq 1, \quad w_i \geq w_{i+1},\ 2 \leq i \leq n-1, \quad w_n \geq 0. \qquad (2)$$

Alternatively, we delta encode weights and obtain *sheared weights* $\vec{v} = [v_2, \dots, v_n]^T \in \mathbb{R}^{n-1}$ with $v_i = w_i - w_{i+1}, i \in \{2, \dots, n-1\}$ and $v_n = w_n$. This shears the simplex's edges emanating from the origin onto the coordinate systems' axes, as shown in Fig. 2c and 2d. Thereby, we get a *sheared simplex* with the following inequalities:

$$\sum_{i=2}^{n} i \cdot v_i \leq 1, \quad v_i \geq 0,\ 2 \leq i \leq n. \qquad (3)$$

We call a tuple inside the respective polyhedral a *valid tuple*. Conversely, a tuple violating those conditions is an *invalid tuple*.

We derive methods for discretizing the simplex. A method is optimal, if two conditions are met: First, samples inside the simplex are distributed uniformly, since, in general, we must assume that all weights are equally likely. Second, the discretization contains no invalid tuples. For most methods, we must relax those goals, but present an optimal one in Sec. 4.6. Further, the sample spacing must be controllable to trade memory space against image quality. Moreover, compression and decompression must be $\mathcal{O}(1)$.

To discretize the tetrahedron, we could use subdivision. However, performance tests revealed that decompression is impractical with $\mathcal{O}(B)$ run-time complexity, where B is the bit count used for weights. With a clustering method, we could replace weights by an index into a LUT that contains the decompressed weights. However, even for small B, LUT sizes become too large to be practical.

4.3. POT AABB (POT)

With the inequalities (2), we see that $0 \leq w_i \leq i^{-1}$ with $2 \leq i \leq n$. Thus, we envelop the simplex with an axis-aligned bounding box (AABB) whose i-th side-length is rounded up to the next power-of-two (POT) of i^{-1} (cf. Fig. 3a). Then, the sample count along each axis is a POT, too. This enables fast weight extraction through bit operations. The resulting sample spacing is $\frac{1/2}{2^{B_2}-1}$ along all axes, where B_2 is the number of bits spent for w_2. As the maximum value of w_i gets smaller with increasing i, the number of bits decreases. For example, $w_4 \leq 1/4$, needs one bit less than $w_2 \leq 1/2$.

By that, we can, however, encode invalid samples for those w_i whose maximum value is not a power-of-two. For example, w_3 can represent values larger than $1/3$, although we never need them. We can avoid that by sampling those w_i more densely. However, this has no effect on the maximum sampling error. In either case, this methods suffers from invalid tuples as shown in Fig. 3a.

4.4. Any AABB (Any)

To reduce invalid tuples, we sample each axis with a non-POT sample count and try to make the total sample count a POT. Therefore, we enclose the simplex with an $(n-1)$-D AABB. Its axis-lengths are $a_i = i^{-1}, 2 \leq i \leq n$. We seek an isotropic grid of $Q = 2^B$ points that covers the AABB entirely. In general, this is not possible for two reasons: First, we will not exactly hit a power-of-two for Q. Therefore, we want to find a number $Q \leq 2^B$ as close as possible to 2^B. Second, the grid can only be almost isotropic and we must sample some directions more densely than others. For each axis, we must determine the number of points q_i such that $\prod_{i=2}^{n} q_i = Q$. The ratio between the number of points and the side lengths must

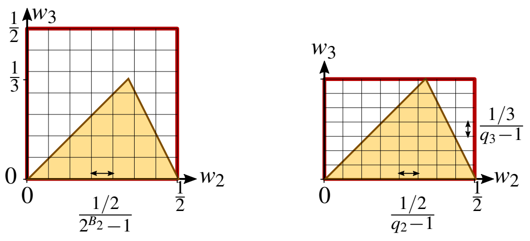


Figure 3: *POT and Any AABB. The triangles show the set of valid tuples. The grid indicates the discretization and the double-sided arrows its sample spacings. (a) We encode weights as points inside an AABB (red box) whose side lengths are power-of-twos. B_2 is the number of bits spent for w_2. (b) Weights are inside an AABB whose side lengths match the maximum value of each weight. q_i is the number of samples per axis.*

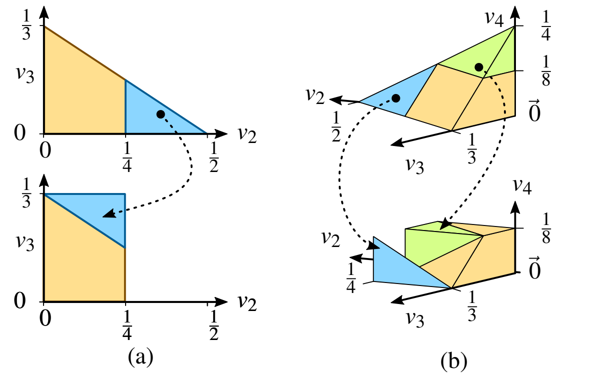


Figure 4: *Simplex Chopping. (a) For three weights, we cut the triangle into two triangles shown in blue and yellow. We transform the triangles to a rectangle avoiding invalid tuples. (b) For four weights, cutting and transforming polytopes leaves invalid tuples.*

be the same for all axes, hence $q_i = \lambda a_i$. We then compute λ:

$$Q = \prod_{i=2}^{n} q_i = \prod_{i=2}^{n} \lambda \cdot a_i = \lambda^{n-1} \prod_{i=2}^{n} a_i \quad \Rightarrow \quad \lambda = \sqrt[n-1]{\frac{Q}{\prod_{i=2}^{n} a_i}}.$$

As $q_i = \lambda a_i$ is usually not integral, we round slightly below or equal the desired amount. From a quantized weight $x_i \in \{0,\dots,q_i - 1\}$, we obtain a unique index $C(\vec{x})$ that we use as the compressed value. For decompression, we use $d_i(C)$:

$$C(\vec{x}) = \sum_{i=2}^{n} x_i \cdot \prod_{j=2}^{i-1} q_j, \qquad d_i(C) = \left\lfloor \frac{C}{\prod_{j=2}^{i-1} q_j} \right\rfloor \bmod q_i.$$

The divisor and module are constant so compilers can replace them by shifts and multiplications. Fig. 3b shows that Any AABB reduces the number of invalid tuples over POT AABB.

4.5. Simplex Chopping

We reduce invalid tuples by chopping the sheared simplex at $v_2 \geq 1/4$ and at $v_4 \geq \frac{1}{8}$. We transform the resulting parts as shown in Fig. 4 to fit an AABB. During sampling, we carefully treat ambiguities along the split planes. Since, we are only able to avoid invalid tuples for three weights, we do not further consider this method.

4.6. Optimal Simplex Sampling (OSS)

OSS is a weight representation that encodes valid tuples, only. We exclude the possibility to encode invalid tuples by design. Our method samples weights uniformly in each axis. Therefore, it is optimal for i.i.d. weight tuples. Our compression and decompression have $\mathcal{O}(1)$ time and space complexity. We derive formulas for four weights, which is sufficient for most practical cases.

Three Weights To only discretize the set of valid weights, we intersect the triangle that contains all possible weights with a 2D Cartesian grid. We make sure that $[w_2, w_3]^T = [0;0]^T$ and $[w_2, w_3]^T = [1/2, 0]^T$ are sampled. N is the number of samples along the bottom row, i.e., where $w_3 = 0$. We assign a *weight index* W to each valid tuple, as indicated by the numbers inside the circles of Fig. 5. The

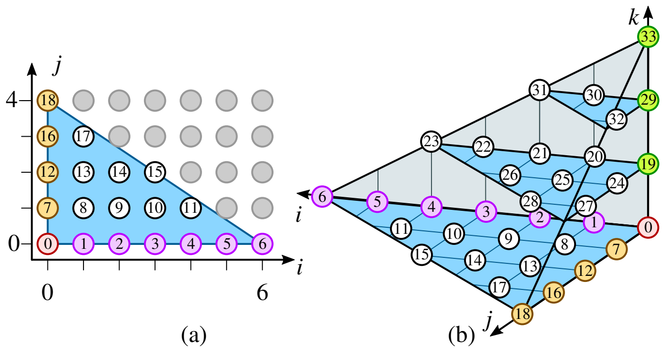


Figure 5: *Assigning Indices for OSS. (a) For the quantized and sheared weights i, j, that correspond to valid weights, we compute a unique index (numbers inside non-grey circles). The grey circles represent invalid tuples. The indices of the red and yellow circles indicate the base indices of each row. (b) For 4D weights, we quantize and sheared weights to indices i, j, k. The red and green circles indicate the base indices of each stacked triangle.*

weight index serves as our compressed weight. Therefore, we quantize and sheared weights $[w_2, w_3]^T \in [0; 1/2] \times [0; 1/3]$ to an index-pair $[i, j]^T$ and compute W:

$$\begin{aligned} W(i,j) &= s_N(j) + i, \\ j = \text{quant}(w_3), &\quad i = \text{quant}(w_2) - j, \\ \text{quant}(w) &= \text{rd}(2w(N-1)), \end{aligned} \tag{4}$$

where $\text{rd}(\cdot)$ rounds to the nearest valid weight component. Thereby, $s_N(j)$ is the *base index of row* j, shown as red and yellow circles in Fig. 5a. Computing $s_N(j)$ iteratively is too slow and we, therefore, derive a closed form as shown in Appx. A:

$$s_N(j) = \left\lfloor -\frac{3}{4}j^2 + \frac{1}{2}j + jN + \frac{1}{4} \right\rfloor = \lfloor \tilde{s}_N(j) \rfloor. \tag{5}$$

In Appx. B, we show that the number of samples per triangle is

$$S(N) = \left\lfloor \frac{1}{3}(N^2 + N + 1) \right\rfloor. \tag{6}$$

This number is important to compute the bit count of 3D weights.

For decompression, we first need to find the row index j_0 from a given index W such that

$$s_N(j_0) \leq W < s_N(j_0 + 1).$$

Solving this problem with binary search would make decompression too slow. Instead, we solve the equivalent root-finding problem $\tilde{s}_N(\tilde{j}_0) = W$. Due to numerical issues, we mirror the function to

$$\tilde{s}_N\left(\frac{2N+1}{3}\right) - \tilde{s}_N(\tilde{j}_0) = \underbrace{\tilde{s}_N\left(\frac{2N+1}{3}\right) - W}_{W'},$$

and obtain the integer root j_0 by flooring the relevant real root $\tilde{j}_0$:

$$j_0 = \lfloor \tilde{j}_0 \rfloor = \left\lfloor \frac{1}{3}\left(2N + 1 + \sqrt{12W'}\right) \right\rfloor.$$

Float arithmetic might introduce off-by-one errors for j_0 if W is near a base index $s_N(j_0)$. We address them with an extra check. Finally, we invert Eqs. (4) to compute w_2 and w_3.

Four Weights For compressing 4D weights $[w_2, w_3, w_4]^T \in [0, 1/2] \times [0, 1/3] \times [0, 1/4]$, we extend the index pair of Eq. (4) to a triple $[i, j, k]^T$ and compute a weight index for four weights U:

$$U(i,j,k) = u(k) + s_{N-2k}(j) + i,$$
$$k = \mathrm{quant}(w_4),\ j = \mathrm{quant}(w_3) - k,\ i = \mathrm{quant}(w_2) - j.$$

U is the compressed representation for four weights. Thereby, $u(k)$ is the *base index of the k-th triangle*. By that, we stack multiple triangles used for 3D weights, with decreasing number of samples at their bottom edges, as shown in Figure 5b. An iterative algorithm for computing $u(k)$ would be too slow. In Appx. C, we derive how to obtain $u(k)$ in $\mathcal{O}(1)$. The base indices are shown by red and green circles in Fig. 5b and are computed as follows:

$$u(k) := \lfloor \tilde{u}(k) \rfloor = \left\lfloor \frac{1}{9} \left[3Nk(N-2k+3) + k(4k^2 - 9k + 6) + 2 \right] \right\rfloor .$$

As the w_2 axis is twice as long as w_4 axis, there are $\left\lfloor \frac{N+1}{2} \right\rfloor$ stacked triangles. Thus, the sample count per tetrahedron is:

$$T(N) := u\left(\left\lfloor \frac{N+1}{2} \right\rfloor \right).$$

We set N such, that $T(N)$ is smaller or equal a power-of-two.

For decompression, we avoid expensive binary search, make it an $\mathcal{O}(1)$ root-finding problem, and mirror the functional to increase numerical stability:

$$u(k_0) \le U < u(k_0+1) \quad \Rightarrow \quad \tilde{u}(\tilde{k}_0) = U$$
$$\Rightarrow \tilde{u}\left(\frac{N+1}{2}\right) - \tilde{u}(k_0) = \underbrace{\tilde{u}\left(\frac{N+1}{2}\right) - U}_{U'} .$$

We find the relevant non-integer root $\tilde{k}_0$

$$\tilde{k}_0 = \frac{1}{4}\left[2N + 3 - \left(c + \frac{1}{c} \right) \right], \text{with}$$
$$c = \sqrt[3]{72U' - 1 + \sqrt{72U'(72U'-2)}} \approx \sqrt[3]{144U'}.$$

The approximation error is smaller than $3.2 \cdot 10^{-6}$ for $U' \ge 1$. Next, we compute $k_0 = \lfloor \tilde{k}_0 \rfloor$ and fix off-by-one errors in case U is close to base index $u(k_0)$. With $W_{k_0} = U - u(k_0)$, we get a three-weight decompression problem $W_{k_0} = s_{N-2k_0}(j) + i$ delivering i_0 and j_0.

4.7. Weight Compression Comparison

In Tab. 2, we list how many bits a weight compression method (first column) requires to be as precise as an uncompressed representation (first row). We measure precision using the maximum sample spacing Δ between two weight components. Consider column 4×16-*bit float*: Uncompressed, we use four 16-bit half precision floats. In row *Float*, we see that this takes up 64 bits. We require 44 bits for four 11-bit fixed-point numbers, to maintain the same maximum quantization error (cf. row *Fixed*). With the *Cube* method, we save one 11-Bit fixed-point number, and use 33 Bit. With *POT* and *Any*, we can further reduce the memory consumption. Finally, OSS performs best requiring 25.8 bits. OSS achieves compression ratios of 1.6:1 to 2.5:1 over the uncompressed representations while still maintaining the same maximum sample spacing.

	4×32-bit float	4×16-bit fixed	4×16-bit float	4×8-bit fixed
Δ	$1/2^{24}$	$1/(2^{16}-1)$	$1/2^{11}$	$1/(2^8-1)$
Float	128	-	64	-
Fixed	96	64	44	32
Cube	72	48	33	24
POT	68	44	29	20
Any	67.4	43.4	28.4	19.4
OSS	64.4	40.8	25.8	16.8

Table 2: *Bits Required to Represent Four Weights at a Sample Spacing of Δ. For each compressed representation (first column), we list the number of bits required to match the precision Δ of an uncompressed representation (first row).*

5. Results and Discussion

In this section, we evaluate our compression techniques. All measurements were taken on an Intel i7-10750H laptop at 2.6 GHz with an Nvidia Quadro T1000 and an Nvidia RTX 2080 eGPU.

5.1. GPU Decompression Timings

We create 1024 instances from 25 individual meshes optimized for vertex cache reuse [SNB07], as shown in Fig. 1. Scenes with comparable geometric complexity that use bone animations can be found in video games [CF15]. Our OpenGL implementation uses a separate draw call for each instance and renders to a 1920×1080 window. The meshes consist of 2k–34k vertices, up to 256 bones, and four weights and indices per vertex. We use 32-bit floats to represent positions, normals, and texture coordinates. The vertex shader carries out weight and index decompression and processes 15M vertices per frame. For linear vertex blending, we upload 54k matrices (ca. 3.3 MiB) per frame. Our index compression yields 10 bpv for index tuples. The per-mesh LUTs are kept in uniform buffers and require 0.2 bpv–2.2 bpv with 1.0 bpv on average (split over the 25 meshes and not 1024 instances).

For compressed vertex-blend attributes, we assign a bit-budget of 32 and 48 bits, respectively. 10 bits for index tuples leaves 22 and 38 bits for weights. Tab. 3 summarizes frame times for different compressed and uncompressed weight representations. Column Δ shows the maximum sample spacing between two weight samples. For comparison, we list timings for uncompressed representations using four 8-bit and 16-bit fixed-point, as well as, 16-bit and 32-bit float weights. Uncompressed indices fit in $4 \times 8 = 32$ bits.

We observe that our compressed representations are almost always significantly faster than uncompressed representations. The only exception is Any AABB with 38-bit weights, which needs 64-bit integer division and modulo operations degrading performance.

	Uncompressed				Compressed							
Method	Float	Fixed	Float	Fixed	Cube	POT	Any	OSS	Cube	POT	Any	OSS
Indices [Bit]	4×8	4×8	4×8	4×8	10	10	10	10	10	10	10	10
Weights [Bit]	4×32	4×16	4×16	4×8	22	22	22	22	38	38	38	38
Total [Bit]	160	96	96	64	32	32	32	32	48	48	48	48
Δ	6.0e−8	1.5e−5	4.9e−4	3.9e−3	7.9e−3	2.6e−3	2.2e−3	1.2e−3	2.4e−4	6.1e−5	5.3e−5	2.9e−5
Quadro T1000 [ms]	17.4	16.8	16.8	16.4	12.9	12.9	12.9	12.9	14.1	14.1	39.8	14.1
RTX 2080 [ms]	4.3	4.2	4.2	4.1	3.4	3.4	3.4	3.4	3.6	3.6	9.6	3.6

Table 3: *GPU Rendering Timings. For* Uncompressed *and* Compressed *methods, we list the memory usage for* Indices *and* Weights. Δ *is the sample spacing between two weight components. For index compression, a uniform buffer for each LUT is required. When splitting the LUT size over the vertices we obtain an additional 1.0 bpv on average, which is not included in the row* Indices. *We provide rendering timings on an Nvidia* Quadro T1000 *and an Nvidia* RTX 2080.

For fair comparisons, we consider methods with similar Δ. 22-bit OSS is faster, more accurate, and more memory efficient than 4×8 fixed-point weights. 38-bit OSS is a more accurate than 4×16 float weights and only slightly less accurate than 4×16 fixed-point weights. At the same time 38-bit OSS is faster and uses only half the size than its uncompressed rivals.

5.2. Image Quality

Cube, POT AABB, and OSS are capable of representing any fixed-point weight representation bit-exactly. Any AABB is not bit exact, but maintains the same maximum sample spacing. Additionally, all methods maintain the same maximum sample spacing of any float format. Therefore, we consider them as visually lossless.

Fig. 6 compares the quality of 4×32-bit float weights (top left) against all methods of Tab. 3 through difference images. A checkerboard texture highlights errors. The 4×16-bit float and fixed representations are visually indistinguishable from 4×32-bit float weights. 4×8-bit fixed representation shows errors along the checkerboard edges. All 22-bit compressed representations, except for *Cube*, visually outperform 4×8-bit fixed. This is in accordance with the smaller sample spacing Δ. 22-bit OSS has the least difference. The difference images for 38-bit show that the image quality is visually indistinguishable from 4×32-bit float weights.

5.3. CPU Compression Timings

Our non-optimized, single-threaded, non-SIMD code compresses ca. 50M four-weight floats to 32-bit words. For comparison, we copy 66M four-weight floats per second. We compress about 45M index tuples per second including hash-map creation. Therefore, we consider our compression fast enough for asset preparation.

5.4. Offline Compression

For additional compression required by streaming and offline-storage scenarios, we use the general-purpose compressor zlib [GA17] configured for best compression. In Tab. 4, we compare bpv statistics for compressed against uncompressed weights and indices across all meshes from Tab 1. We use 41-bit for OSS to match the precision of 4×16-bit uncompressed fixed-point weights. Without our compression, zlib compresses to an average compression ratio of 4.1:1, whilst with our methods, we improve the average compression ratio to 5:1.

6. Conclusion and Future Work

In this paper, we introduce techniques for compressing vertex-blend attributes used for skeleton-based animations. By exploiting coherence amongst bone indices, we compress them to 2.3:1–3.5:1. We observe that weights describe an $(n-1)$-D simplex and provide several sampling methods. With OSS, we propose an optimal sampling strategy achieving compression ratios between 1.6:1 and 2.5:1 while being at least visually lossless. Combining our bone index and weight compression, we speed frame times, require less memory, and obtain a similar image quality over uncompressed representations. For lossy compression, the controllable error of our representations allows even higher compression ratios, while being able to control visual quality. Moreover, our representations compress well with general purpose codecs. Since compression and decompression is in $\mathcal{O}(1)$, our methods are useful for many applications. Decompression runs in a standard vertex shader with 32-bit arithmetic. The supplemental material contains code for OSS.

In the future, we will pair our methods with meshlets. They are flexible enough to allow a variable weight count per vertex and

Indices & Weights	ZLib	Minimum [bpv]	Avgerage [bpv]	Maximum [bpv]
Uncompressed	No	96.0	96.0	96.0
Uncompressed	Yes	11.6	23.7	38.7
Compressed	No	50.1	52.4	55.0
Compressed	Yes	9.3	19.2	30.1

Table 4: *Offline Compression Result. For uncompressed weights, we use* 4×16*-bit uncompressed fixed-point weights and for compressed weights the slightly more accurate* 41*-bit OSS weight compression (cf. Tab. 2). Uncompressed indices are represented with* $4 \times \log_2(B)$ *bits. For compressed indices, we use the method from Sec. 3 and we split the LUT size across the vertices.*

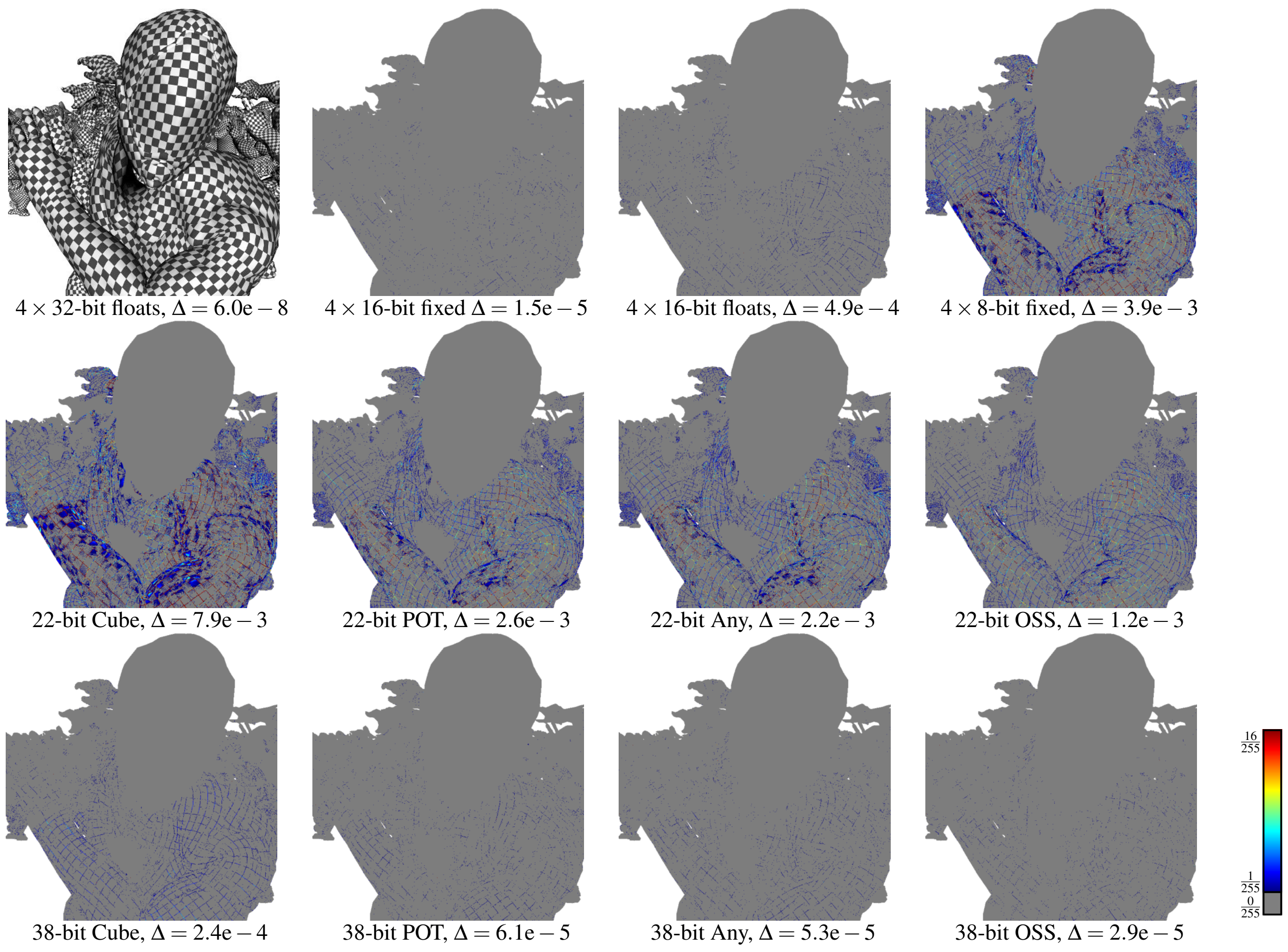

4×32-bit floats, $\Delta = 6.0e-8$
4×16-bit fixed $\Delta = 1.5e-5$
4×16-bit floats, $\Delta = 4.9e-4$
4×8-bit fixed, $\Delta = 3.9e-3$
22-bit Cube, $\Delta = 7.9e-3$
22-bit POT, $\Delta = 2.6e-3$
22-bit Any, $\Delta = 2.2e-3$
22-bit OSS, $\Delta = 1.2e-3$
38-bit Cube, $\Delta = 2.4e-4$
38-bit POT, $\Delta = 6.1e-5$
38-bit Any, $\Delta = 5.3e-5$
38-bit OSS, $\Delta = 2.9e-5$
$\frac{16}{255}$
$\frac{1}{255}$
$\frac{0}{255}$

Figure 6: *Quality Comparison. We obtain difference images when comparing renderings using 4×32-bit float weights (top left) against uncompressed representations (top row), compressed representations with 22 bits (middle row), and 38 bits (bottom row) for four weights, respectively. The color-scale on the right encodes the L^∞-norm of the pixel differences using normalized color intensities.*

we expect additional compression benefits. Further the unique index tuple set per meshlet is likely to be much smaller than for an entire mesh. Further, for OSS, we will support more weights for cage-based animations, blend-shapes, and local barycentric coordinates [ZDL*14]. Early experiments with automated derivation of base index formulas and numerical root-finding indicate that OSS could scale well with more than four weights.

Acknowledgements

We thank C. Eisenacher, M. Chajdas, and G. Greiner for valuable comments and Mixamo for meshes in Figs. 1, 6, and Tab. 1.

References

[AG05] ALLIEZ P., GOTSMAN C.: Recent Advances in Compression of 3D Meshes. In *Advances in Multiresolution for Geometric Modelling* (2005), Springer. 2

[Ale02] ALEXA M.: Linear Combination of Transformations. *ACM Trans. Graph. 21*, 3 (July 2002), 380–387. 3

[Bin20] BINOMIAL LLC: Advanced DXTn texture compression library, 2020. URL: https://github.com/BinomialLLC/crunch. 2

[Bin21] BINOMIAL LLC: Basis Universal Supercompressed GPU Texture Codec, 2021. URL: https://github.com/BinomialLLC/basis_universal. 2

[Bre16] BRENNAN C.: Getting the Most out of Delta Color Compression, 2016. URL: https://gpuopen.com/learn/dcc-overview/. 2

[CDE*14] CIGOLLE Z. H., DONOW S., EVANGELAKOS D., MARA M., MCGUIRE M., MEYER Q.: A Survey of Efficient Representations for Independent Unit Vectors. *Journal of Computer Graphics Techniques (JCGT) 3*, 2 (April 2014). 2

[CF15] COURNOYER F., FORTIER A.: Massive Crowd on Assassin's Creed Unity: AI Recycling, 2015. URL: https://www.gdcvault.com/play/1022141/Massive-Crowd-on-Assassin-s. 6

[Col19] COLLET Y.: *LZ4 Block Format Description*, 2019.

URL: https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md. 2

[Cry19] CRYTEK GMBH: CRYENGINE V Manual, 2019. URL: https://docs.cryengine.com/display/CEMANUAL/Biped+Rigging. 3

[Dee95] DEERING M.: Geometry Compression. SIGGRAPH '95. 2

[FH11] FREY I. Z., HERZEG I.: Spherical Skinning with Dual Quaternions and QTangents. In *ACM SIGGRAPH 2011 Talks* (2011), SIGGRAPH '11, Association for Computing Machinery. 1

[Fré17] FRÉCHETTE N.: Simple and Powerful Animation Compression, 2017. URL: https://www.gdcvault.com/play/1024009/Simple-and-Powerful-Animation. 3

[GA17] GAILLY J.-L., ADLER M.: zlib 1.2.11.1, 2017. URL: https://zlib.net/. 7

[Gar19] GARRAD A.: *Khronos Data Format Specification v1.3.1*. The Khronos Group Inc., 2019. 2

[GGW20] GEFFROY J., GNEITING A., WANG Y.: Rendering the hellscape of doom eternal. In *ACM SIGGRAPH '20: ACM SIGGRAPH 2020 Courses* (2020), SIGGRAPH '20, Association for Computing Machinery. 1

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: Efficient Depth Buffer Compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (New York, NY, USA, 2006), GH '06, Association for Computing Machinery, p. 103–110. 2

[JBG17] JAKOB J., BUCHENAU C., GUTHE M.: A Parallel Approach to Compression and Decompression of Triangle Meshes Using the GPU. *Comput. Graph. Forum 36*, 5 (Aug. 2017). 2

[KCvO08] KAVAN L., COLLINS S., ŽÁRA J., O'SULLIVAN C.: Geometric Skinning with Approximate Dual Quaternion Blending. *ACM Trans. Graph. 27*, 4 (Nov. 2008). 2, 3

[Khr21] THE KHRONOS GROUP INC.: *Vulkan 1.2.169 - A Specification*, 2021. 2

[KISS15] KEINERT B., INNMANN M., SÄNGER M., STAMMINGER M.: Spherical Fibonacci Mapping. *ACM Trans. Graph. 34*, 6 (Oct. 2015). 2

[KPM16] KRAJCEVSKI P., PRATAPA S., MANOCHA D.: GST: GPU-Decodable Supercompressed Textures. *ACM Trans. Graph. 35*, 6 (2016). 2

[Kub20] KUBISCH C.: Using Mesh Shaders for Professional Graphics, 2020. URL: https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/. 2

[Kv05] KAVAN L., ŽÁRA J.: Spherical Blend Skinning: A Real-Time Deformation of Articulated Models. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), I3D '05, Association for Computing Machinery, p. 9–16. 3

[KXW*18] KWAN K. C., XU X., WAN L., WONG T., PANG W.: Packing Vertex Data into Hardware-Decompressible Textures. *IEEE Transactions on Visualization and Computer Graphics 24*, 5 (2018). 2

[LCL10] LEE J., CHOE S., LEE S.: Compression of 3D Mesh Geometry and Vertex Attributes for Mobile Graphics. *JCSE 4* (09 2010). 2

[LDJ*19] LUO G., DENG Z., JIN X., ZHAO X., ZENG W., XIE W., SEO H.: 3D Mesh Animation Compression Based on Adaptive Spatio-Temporal Segmentation. I3D '19, Association for Computing Machinery. 3

[Mic20] MICROSOFT: DirectX-Specs Mesh Shader v0.85, 2020. URL: https://microsoft.github.io/DirectX-Specs/d3d/MeshShader.html. 2

[MJT14] MCALLISTER D. K., JOLY A., TONG P.: Lossless Frame Buffer Color Compression, 2014. 2

[MKSS12] MEYER Q., KEINERT B., SUSSNER G., STAMMINGER M.: Data-Parallel Decompression of Triangle Mesh Topology. *Computer Graphics Forum 31*, 8 (2012). 2

[MLDH15] MAGLO A., LAVOUÉ G., DUPONT F., HUDELOT C.: 3D Mesh Compression: Survey, Comparisons, and Emerging Trends. 2

[MSGS11] MEYER Q., SUSSNER G., GREINER G., STAMMINGER M.: Adaptive Level-of-Precision for GPU-Rendering. In *Vision, Modeling, and Visualization (2011)* (2011), The Eurographics Association. 2

[MSS*10] MEYER Q., SÜSSMUTH J., SUSSNER G., STAMMINGER M., GREINER G.: On Floating-Point Normal Vectors. In *Proceedings of the 21st Eurographics Conference on Rendering* (2010), EGSR'10, Eurographics Association. 2

[MTLT89] MAGNENAT-THALMANN N., LAPERRIÈRE R., THALMANN D.: Joint-Dependent Local Deformations for Hand Animation and Object Grasping. In *Proceedings on Graphics Interface '88* (CAN, 1989), Canadian Information Processing Society, p. 26–33. 2, 3

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive Scalable Texture Compression. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (2012), Eurographics Association. 2

[NVB*20] NOORDSIJ L., VLUGT S., BAMAKHRAMA M., AL-ARS Z., LINDSTROM P.: Parallelization of Variable Rate Decompression through Metadata. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2020), pp. 245–252. 2

[OBGB11] OLANO M., BAKER D., GRIFFIN W., BARCZAK J.: Variable Bit Rate GPU Texture Decompression. In *Proceedings of the Twenty-Second Eurographics Conference on Rendering* (2011), Eurographics Association. 2

[PBCK05] PURNOMO B., BILODEAU J., COHEN J. D., KUMAR S.: Hardware-Compatible Vertex Compression Using Quantization and Simplification. HWWS '05, Association for Computing Machinery. 2

[Per12] PERSSON E.: Creating vast game worlds: Experiences from avalanche studios. In *ACM SIGGRAPH 2012 Talks* (2012), SIGGRAPH '12, Association for Computing Machinery. 1

[PKJ05] PENG J., KIM C.-S., JAY KUO C.-C.: Technologies for 3D Mesh Compression: A Survey. *Journal of Visual Communication and Image Representation 16*, 6 (2005). 2

[PLS12] POOL J., LASTRA A., SINGH M.: Lossless Compression of Variable-Precision Floating-Point Buffers on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2012), I3D '12, Association for Computing Machinery. 2

[RB20] ROUSSEAU S., BOUBEKEUR T.: Unorganized Unit Vectors Sets Quantization. *Journal of Computer Graphics Techniques (JCGT) 9*, 3 (2020). 2

[SLK20] SAKHARNYKH N., LASALLE D., KARSIN B.: Optimizing Data Transfer Using Lossless Compression with NVIDIA nvcomp, 2020. URL: https://developer.nvidia.com/blog/optimizing-data-transfer-using-lossless-compression-with-nvcomp/. 2

[SLY20] SEILER L., LIN D., YUKSEL C.: Compacted CPU/GPU Data Compression via Modified Virtual Address Translation. *Proc. ACM Comput. Graph. Interact. Tech. 3*, 2 (Aug. 2020). 2

[SNB07] SANDER P. V., NEHAB D., BARCZAK J.: Fast Triangle Reordering for Vertex Locality and Reduced Overdraw. *ACM Trans. Graph. 26*, 3 (July 2007), 89–es. 6

[SSK05] SATTLER M., SARLETTE R., KLEIN R.: Simple and Efficient Compression of Animation Sequences. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005). 3

[SWR*08] STRÖM J., WENNERSTEN P., RASMUSSON J., HASSELGREN J., MUNKBERG J., CLARBERG P., AKENINE-MÖLLER T.: Floating-Point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware* (2008). 2

[Uni21] UNITY: QualitySettings.skinWeights, 2021. URL: https://docs.unity3d.com/2021.1/Documentation/ScriptReference/QualitySettings-skinWeights.html. 3

[ZDL*14] ZHANG J., DENG B., LIU Z., PATANÈ G., BOUAZIZ S., HORMANN K., LIU L.: Local Barycentric Coordinates. 8

Appendix A: Base Index for Three Weights

We derive the base index $s_N(j)$ of Eq. (5) shown in the in the red and yellow circles of Fig. 5a. The number of indices per row j is

$$l(j) = \left\lfloor \frac{2N-3i}{2} \right\rfloor$$

and the total number of rows is

$$J(N) = \lfloor \tilde{J}(N) \rfloor = \left\lfloor \frac{2N+1}{3} \right\rfloor = \begin{cases} \frac{2}{3}N & \text{if } N \equiv 0 \bmod 3 \\ \frac{2}{3}N + \frac{1}{3} & \text{if } N \equiv 1 \bmod 3 \\ \frac{2}{3}N - \frac{1}{3} & \text{if } N \equiv 2 \bmod 3. \end{cases} \quad (7)$$

The base index of each row is

$$s_N(j) = \sum_{i=0}^{j-1} l(j) = \sum_{i=0}^{j-1} \left\lfloor \frac{2N-3i}{2} \right\rfloor$$

Since $\lfloor \frac{a}{b} \rfloor = \frac{1}{b}(a - a \bmod b)$, we get

$$\begin{aligned} s_N(j) &= \frac{1}{2}\left(\sum_{i=0}^{j-1} (2N-3i) - \sum_{i=0}^{j-1} \underbrace{(2N-3i) \bmod 2}_{0,1,0,1,\ldots} \right) \\ &= -\frac{3}{4}j^2 + \frac{3}{4}j + jN - \underbrace{\frac{1}{2}\left\lfloor \frac{j}{2} \right\rfloor}_{\frac{0}{2},\frac{0}{2},\frac{1}{2},\frac{1}{2},\frac{2}{2},\frac{2}{2},\ldots} \\ &= \left\lfloor -\frac{3}{4}j^2 + \frac{1}{2}j + jN + \frac{1}{4} \right\rfloor = \lfloor \tilde{s}_N(j) \rfloor \quad (8) \\ &= -\frac{3}{4}j^2 + \frac{1}{2}j + jN + \Gamma \text{ with } \Gamma = \begin{cases} \frac{1}{4}, & j \text{ is odd} \\ 0, & \text{else.} \end{cases} \quad (9) \end{aligned}$$

Eq. (8) suits code, Eq. (9) derivations, and $\tilde{s}_N(j)$ root finding.

Appendix B: Number of Indices per Triangle

For the triangle sample count $S(N) := s_N(J(N))$ of Eq. (6), we have three cases, from which we infer the cases of Eq. (9).

Case 1: $N \equiv 0 \bmod 3 \Rightarrow J \equiv 0 \bmod 2$

$$s_N(J(N)) = -\frac{3}{4}\left(\frac{2}{3}N\right)^2 + \frac{1}{2}\left(\frac{2}{3}N\right) + \left(\frac{2}{3}N\right)N = \frac{1}{3}N^2 + \frac{1}{3}N.$$

Case 2: $N \equiv 1 \bmod 3 \Rightarrow J \equiv 1 \bmod 2$

$$\begin{aligned} s_N(J(N)) &= -\frac{3}{4}\left(\frac{2}{3}N + \frac{1}{3}\right)^2 + \frac{1}{2}\left(\frac{2}{3}N + \frac{1}{3}\right) + \left(\frac{2}{3}N + \frac{1}{3}\right)N + \frac{1}{4} \\ &= \frac{1}{3}N^2 + \frac{1}{3}N + \frac{1}{3}. \end{aligned}$$

Case 3: $N \equiv 2 \bmod 3 \Rightarrow J \equiv 1 \bmod 2$

$$\begin{aligned} s_N(J(N)) &= -\frac{3}{4}\left(\frac{2}{3}N - \frac{1}{3}\right)^2 + \frac{1}{2}\left(\frac{2}{3}N - \frac{1}{3}\right) + \left(\frac{2}{3}N - \frac{1}{3}\right)N + \frac{1}{4} \\ &= \frac{1}{3}N^2 + \frac{1}{3}N. \end{aligned}$$

We condense these cases to

$$\begin{aligned} S(N) := s_N(J(N)) &= \begin{cases} \frac{1}{3}N^2 + \frac{1}{3}N + \frac{1}{3} & \text{if } N \equiv 1 \bmod 3 \\ \frac{1}{3}N^2 + \frac{1}{3}N & \text{else.} \end{cases} \\ &= \left\lfloor \frac{1}{3}\left(N^2 + N + \tau\right) \right\rfloor, \text{with } \tau \in [1,3]. \end{aligned}$$

Appendix C: Base Index for Four Weights

For 4D weights, we stack triangles and add their respective sample count (cf. Fig. 5b). The k-th triangle has $N-2k$ samples at its bottom row and total of $S(N-2k)$ samples. Summing them up yields the base index of the k-th triangle (red and green circles in Fig. 5b):

$$\begin{aligned} u(k) &= \sum_{i=0}^{k-1} S(N-2i) = \sum_{i=0}^{k-1} \left\lfloor \frac{1}{3} \underbrace{\left(N^2 - 4Ni + 4i^2 + N - 2i + 1\right)}_{\omega} \right\rfloor \\ &= \frac{1}{3}\sum_{i=0}^{k-1} \omega - \underbrace{\frac{1}{3}\sum_{i=0}^{k-1} \underbrace{\omega \bmod 3}_{r_N(k)}}_{\bar{r}_N(k)} \\ &= \frac{1}{9}\left[3N^2k - 6Nk^2 + 9Nk + 4k^3 - 9k^2 + 6k - 3\bar{r}_N(k)\right]. \quad (10) \end{aligned}$$

We can show that

$$r_N(k) = \begin{cases} 1,0,1;\ 1,0,1;\ 1,0,1;\ \ldots, & \text{if } N \equiv 0 \bmod 3 \\ 0,1,1;\ 0,1,1;\ 0,1,1;\ \ldots, & \text{if } N \equiv 1 \bmod 3 \\ 1,1,0;\ 1,1,0;\ 1,1,0;\ \ldots, & \text{if } N \equiv 2 \bmod 3, \end{cases}$$

$$\begin{aligned} \bar{r}_N(k) &= \begin{cases} 0,1,1;\ 2,3,3;\ 4,5,5;\ \ldots, & \text{if } N \equiv 0 \bmod 3 \\ 0,0,1;\ 2,2,3;\ 4,4,5;\ \ldots, & \text{if } N \equiv 1 \bmod 3 \\ 0,1,2;\ 2,3,4;\ 4,5,6;\ \ldots, & \text{if } N \equiv 2 \bmod 3 \end{cases} \\ &= \left\lfloor \frac{2k+\nu_N}{3} \right\rfloor, \nu_N = \begin{cases} 1 & \text{if } N \equiv 0 \bmod 3 \\ 0 & \text{if } N \equiv 1 \bmod 3 \\ 2 & \text{if } N \equiv 2 \bmod 3. \end{cases} \end{aligned}$$

To avoid flooring, we write

$$\begin{aligned} \bar{r}_N(k) &= \frac{2k + \nu_N - (2k+\nu_N) \bmod 3}{3} = \frac{2k + \eta(k)}{3}, \\ \eta(k) &= \begin{cases} 1 - (2k+1) \bmod 3 & \text{if } N \equiv 0 \bmod 3 \\ 2k \bmod 3 & \text{if } N \equiv 1 \bmod 3 \\ 2 - (2k+2) \bmod 3 & \text{if } N \equiv 2 \bmod 3. \end{cases} \end{aligned}$$

Inserting into Eq. (10) yields:

$$\begin{aligned} u(k) &= \frac{1}{9}\left[3N^2k - 6Nk^2 + 9Nk + 4k^3 - 9k^2 + 6k + \eta(k)\right] \\ &= \frac{1}{9}Y + \frac{1}{9}\eta(k). \end{aligned}$$

Note that $\eta(k) \in \{-2;\ldots;2\}$ and Y is integer. $u(k)$ must be integer, so $\frac{1}{9}\eta(k)$ serves as corrective to make $\frac{1}{9}Y$ integer: If $\eta(k) = 2$, then $\frac{1}{9}Y$ is $\frac{2}{9}$ away from being integer, if $\eta(k) = 1$, then $\frac{1}{9}Y$ is $\frac{1}{9}$ away from being integer, etc. All five cases can be subsumed under

$$u(k) = \left\lfloor \frac{1}{9}Y + \frac{2}{9} \right\rfloor = \left\lfloor \frac{1}{9}\left[3Nk(N-2k+3) + k(4k^2 - 9k + 6) + 2\right] \right\rfloor.$$