HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg

Fakultät Elektrotechnik und Informatik

Studiengang: Informatik (B.Sc.)

Bachelorarbeit

Komprimierung von Vertex-Blending-Attributen

Bastian Kuth

Abgabe der Arbeit: 29.01.2021

Betreut durch:

Prof. Dr.-Ing. Quirin Meyer, Hochschule Coburg

Zusammenfassung

Skelettbasierte Animation ist eine Standardmethode der Computergrafik für das Animieren von Spielfiguren. Um ein Mesh animierbar zu machen, benötigt es pro Vertex sogenannte „Vertex-Blending-Attribute“, die den Einfluss eines Skelettknochens auf den jeweiligen Vertex beschreiben. Ein einzelnes Vertex-Blending-Attribut besteht aus einem Gewichtstupel und einem Bone-Index-Tupel gleicher Dimension (z. B. 4 Gewichte und 4 Indizes). Durch den kontinuierlichen Anstieg der Vertexanzahl von Meshes und dem daraus resultierenden Anstieg der Datenmenge entsteht die Notwendigkeit für Kompression. Es existieren allgemeine Ansätze zur Kompression von Vertex-Attributen. Diese erschöpfen allerdings nicht im Speziellen das Kompressionspotenzial von Vertex-Blending-Attributen. Im Rahmen dieser Arbeit wird dieses Potential erforscht und Methoden zur Bone-Index-Kompression und optimalen Quantisierung von gleichmäßig verteilten Blend-Gewichten vorgestellt. Alle gezeigten Methoden entpacken Daten unabhängig für jeden Vertex, wodurch sie direkt in einem Vertex-Shader genutzt werden können. Für beide Attribute werden verlustfreie Kompressionsraten zwischen 2:1 und 2,5:1 erreicht. Weiterhin wird gezeigt, dass bei bandbreitenlimitierten Anwendungen die Nutzung der gezeigten Kompressionsmethoden einen Laufzeitvorteil bringen kann.

Abstract

Skeleton-based animation is a standard technique for animating characters in computer graphics. It requires per-vertex attributes, called vertex-blending attributes, that describe the influence skeleton-bones have on each vertex. A single vertex-blending attribute consists of a weight tuple and a bone index tuple of equal dimensionality (e.g., 4 weight and 4 indices). With the ever-increasing number of vertices and the resulting increase of data, combined with hardware limitations, comes the need for compression. While general approaches for vertex-attribute compression exist, we could not find any techniques that specifically address the compression potential of vertex-blending attributes. In this thesis, we explore this potential and present methods for bone index compression and optimal quantization for uniformly distributed blend-weights. All presented techniques unpack the data independently for each vertex, making them directly usable in a vertex shader. We show that we can reach a lossless compression ratio between 2:1 and 2.5:1 for both attributes. Additionally, we show that our techniques also can speed up rendering if the application is bandwidth limited.

Content

List of Figures

List of tables

1 Introduction

1.1 Motivation

Real-time computer graphics applications continue to push the boundaries of computer hardware. While GPUs reach higher compute power levels with every new generation, the GPUs memory performance grows much slower, often being the limiting factor. Therefore, modern GPUs come with support for compressed textures, which usually are the dominant factor for the GPUs memory usage. Though, with the ever-growing number of vertices per scene, the attributes of a vertex require compression as well. A subset of these attributes are the vertex-blending attributes. To be animatable with the skeleton-based animation technique, meshes require *vertex-blending attributes*. These attributes consist of an n-tuple of indices, each referencing a bone transformation. The other part is an n-tuple of weights, used to blend these transformations together to receive the animated position of the corresponding vertex. In this thesis we present methods for compressing these vertex-blending attributes.

1.2 Related Work

Although vertex-blending attributes make up a large part of a meshes attribute size, it has not yet been explored how to efficiently compress them. Several methods for animation compression such as [Sattler 2005], [Frechette 2017] and [Luo 2019] exist, with all of them focusing on compression of motion-describing data (e.g., keyframes). [Purnomo 2005] quantizes vertex data in general by minimizing a visual error metric without analyzing the individual compressible properties for each attribute. Similarly, the method presented in [Meyer 2011] uses quantized vertex positions instead of level-of-detail for meshes that are positioned distant from the camera to speed up rendering. [Meyer 2010] describes a method for lossless compression of a vertex normal into 51 bits while retaining the precision of float32s. [Frey 2011] suggests storing vertex tangents in quaternion representation to better fit quaternion skinning techniques with the additional benefit of compression.

2 Fundamentals

In this section, we discuss fundamental knowledge required to understand the described techniques of this thesis.

2.1 Skeleton-based Animation

Skeleton-based animation is a method of transforming a mesh (e.g., a character) to perform gestures. To prepare a mesh for animation, an artist must define a skeleton and the corresponding vertex-blending attributes (also called skinning attributes). The *skeleton*, also referred to as *armature* or *rig*, consists of a hierarchical ordered set of bones. This hierarchy fulfills the tree or forest properties from graphing theory. Figure 1 shows an example of such a bone hierarchy: The “pelvis” of the humanoid mesh is the root bone. Attached to the root are the lower end of the spine (“spine_01”) and both femur bones (“thight_r” and “thight_l”). Going down the right leg, the child of “thight_r” is “calf_r” whose child is “foot_r” and so on.

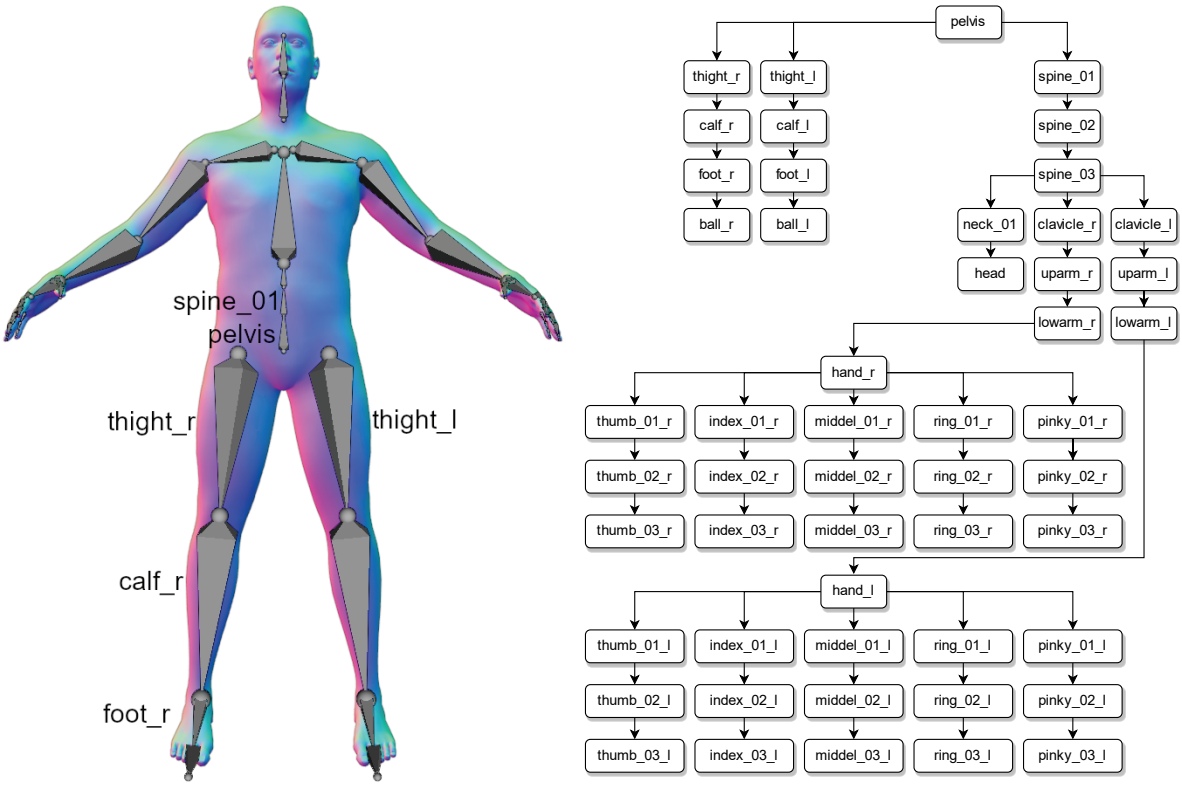
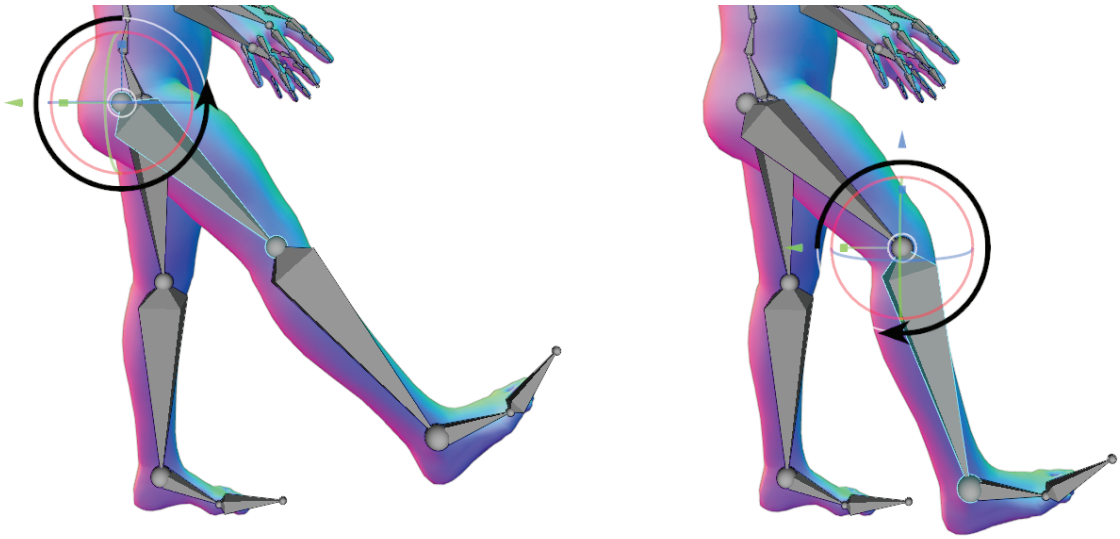


Fig. 1: skeleton hierarchy example

For each bone, we assign a transformation, e.g., a matrix or (dual) quaternion. A bone transformation affects its child bones' transformations. An example of this can be seen in figure 2: The artist wants the humanoid to take a step. She first rotates "thight_r" upwards, rotating all child bones with it (a). Second, she rotates "calf_r" downwards again, resulting in the desired pose (b).



(a) (b)

Fig. 2: animating a step pose

To create an animation from multiple poses, the concept of *keyframes* is introduced. A keyframe describes the transformation, typically a combination of rotations, translations, and scale of a bone, at a given time. Intermediate frames, i.e., frames between two successive keyframes, interpolate transformations from keyframe transformations.

For every vertex, we assign *vertex-blending* or *skinning attributes*. A skinning attribute consists of an n-tuple of *weights* and an n-tuple of bone *indices*. Bone indices reference a transformation assigned to a bone. Each bone index has an associated weight which determines the influence of the respective bone transformation. The maximum number of bone influences per vertex varies between implementations, with 4 being standard for real time rendering applications [Khronos 2016] [Crytek 2015] [Unity 2017]. Figure 3 visualizes the influence the bone "Nose Tip" has on every vertex. Vertices that are only influenced by that bone are marked red. If they are also influenced by other bones, the color will fade to blue with dark blue meaning "no influence".

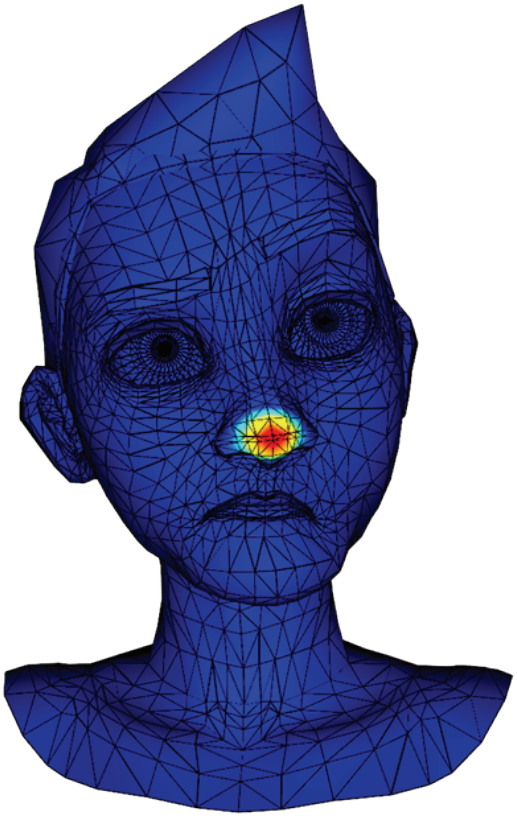


Fig. 3: bone influence visualization

2.2 Geometric Skinning Techniques

Various techniques for blending multiple bone transforms together exist. This section will describe the two most common ones, which are also the methods tested with the compression techniques described in this thesis.

2.2.1 Linear Blend Skinning (LBS)

With a given transformation matrix tuple $T = (t_1, t_2, t_3, \ldots, t_m)$, an index tuple $(I_1, I_2, I_3, \ldots, I_n)$, a weight tuple $(w_1, w_2, w_3, \ldots, w_n)$, and a vertex position v with $m \geq n$, the animated position v' of that vertex is:

$$v' = \sum_{i=1}^{n} w_i \cdot t_{I_i} \cdot v.$$

While being simple to understand and implement, this method suffers from an artifact referred to as the "candy-wrapper" [Kavan 2008]. When rotating one of the bones of a joint by 180 degrees, a vertex that is influenced by both bones will move to the center of rotation. This leads

to unintentional shrinkage of a meshes volume around joints. Figure 4 shows this artifacts appearance.

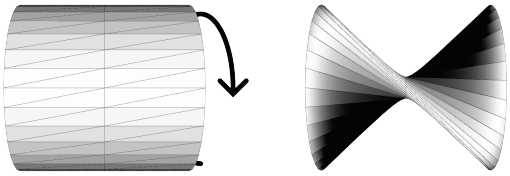


Fig. 4: candy-wrapper artifact using LBS

2.2.2 Dual Quaternion Skinning

Dual Quaternion Skinning (*DQS*), which was first introduced by [Kavan 2008], utilizes dual quaternions when blending transformations together, preserving the volume of the mesh and preventing the "candy-wrapper" artifact. As a unit quaternion represents a rotation only, a dual part is introduced to also represent translations [Kenwright 2012]. Figure 5 shows the same model as in the previous section's figure but using DQS. As can be seen, this method preserves the volume of the mesh near joints when rotating them. The artifact is resolved.

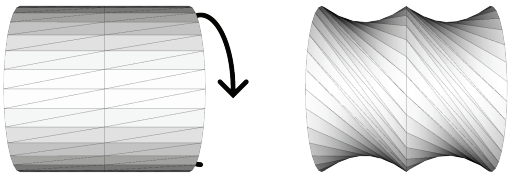


Fig. 5: candy-wrapper artifact using DQS

2.3 Data Quantization

Quantization is used to map values from a bigger set (or even continuous values) to a smaller set by rounding multiple values into one [Lloyd 1982]. Additionally, quantization can make use of known data ranges (minima and maxima). For example, blend weights can only have values between 0 and 1. When saving these values in a float16 or float32, a lot of states of this floating-point number remain unused and therefore wasted.

2.3.1 Quantization in a Single Dimension

In the example shown in figure 6, values between a minimum m and maximum M are quantized into 4 sample points (2 bits). When quantizing, a value gets rounded to its closest sample point. If the original value is exactly between the two closest points, the resulting value will be off by the maximum quantization error, which equals half of the distance between 2 consecutive points or the distance between an extreme and its closest point.

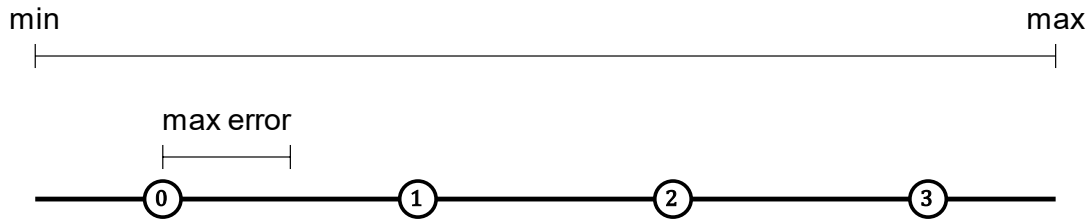


Fig. 6: quantization system

To calculate the maximum error E from the number of sample points P or the number of bits B, we use the following formula:

$$E = \frac{M - m}{2 \cdot P} \text{ or } \frac{M - m}{2^{B+1}}.$$

To quantize a value x to an index I_x:

$$I_x = \min\left(\left\lfloor \frac{x - E}{2 \cdot E} \right\rceil, P - 1\right).$$

And to get the faulty value x':

$$x' = E + 2 \cdot E \cdot I_x = E \cdot (1 + 2 \cdot I_x).$$

2.3.2 Quantization in Multiple Dimensions

When quantizing n-dimensional values ($x \in \mathbb{R}^n$), the indices for each axis are calculated separately and later combined into a single value. The total number of sample points P is split into a smaller number of points p_i for each axis i with

$$\prod_{i=1}^{n} p_i = P.$$

We define the *length* of an axis as its maximum possible value subtracted by its minimum possible value ($M - m$). If the length is the same for every axis (values form a unit cube) the number of sample points per axis p is given by:

$$p = \sqrt[n]{P}.$$

If the length is not the same for every axis, we can assure a uniform distribution of sample points like the following: The "number of points" ratio between two axes must be equal to the length ratio between these axes. For example, for an x-axis with values between 0 and 2 and a y-axis with values between 0 and 1, the value range ratio is 2:1. With a total number of 8 points, the uniform distribution would be 4 on the x-axis and 2 on the y-axis. As we can scale the axis length vector $[l_1, l_2, \dots, l_n]$ with λ to receive the "number of points" vector $[p_1, p_2, \dots, p_n]$, it is given that:

$$\lambda \cdot l_i = p_i \; \forall \; i \leq n$$

$$\prod_{j=1}^{n} (\lambda \cdot l_j) = P \quad \Rightarrow \quad \lambda^n \cdot \prod_{j=1}^{n} (l_j) = P \quad \Rightarrow \quad \lambda = \sqrt[n]{\frac{P}{\prod_{j=1}^{n} (l_j)}}.$$

Therefore, the number of points p_i for an axis i is given by:

$$p_i = \sqrt[n]{P} \cdot \frac{l_i}{\sqrt[n]{\prod_{j=1}^{n} l_j}}.$$

As this term usually does not give integer values for the number of points per axis, rounding is required. Depending on the needs, it can be valid to round every axis' points to the nearest integer. When we do so, it is likely that we get a total number of sample points slightly above or below the desired value, while we also approximately preserve the length ratio of every axis. For compression, it is more suitable to use as many of the available states as possible. Therefore, in this thesis, we will round the values off or up so that we receive a total number of sample points that is slightly below or equal to the desired amount ($\prod_{i=1}^{n} p_i \leq P$).

By taking the formula from the previous section and the just determined number of points per axis, we quantize a value separately for each axis. Subsequently, we combine the indices into a single value $\mathbb{I} \in [0, P)$:

$$\mathbb{I} = \sum_{i=1}^{n} \left(I_i \cdot \prod_{j=1}^{i-1} (p_j) \right).$$

For decoding, the indices can be separated again with:

$$I_i = \left\lfloor \frac{\mathbb{I}}{\prod_{j=1}^{i-1} (p_i)} \right\rfloor \bmod p_i.$$

As the divisor and module of the previous formula stay the same when quantizing multiple values, it might be beneficial to use the invariant integer divisor optimizations described in [Granlund 1994], depending on the hardware used. Alternatively, it is possible to choose a power-of-two as points per axis, so that the division can be replaced by a shift and the modulo with a bitwise AND.

3 Compressing the Indices

Every vertex of our test meshes has a tuple of indices, each pointing at a bone transformation. The number of bits needed to save a single index depends on the number of bones of the mesh. For up to 119 bones from our test meshes taken from table 1, we need $\lceil \log_2(119) \rceil = 7$ bits per index or 28 bits for a complete 4-tuple of indices. In the upcoming section we will present a method to compress this tuple. Section 3.2 and 3.3 will address techniques to reduce the total data size even further.

3.1 Index Tuple Set

Figure 7 visualizes the different index tuples of various models by giving each unique tuple a unique color. As can be seen, regions of vertices with the same tuple appear. For example, in (a) vertices located next to the center of the forearm only have an index to the forearm bone. Vertices located near the wrist have an index to a hand bone as well as to the forearm bone and so on.

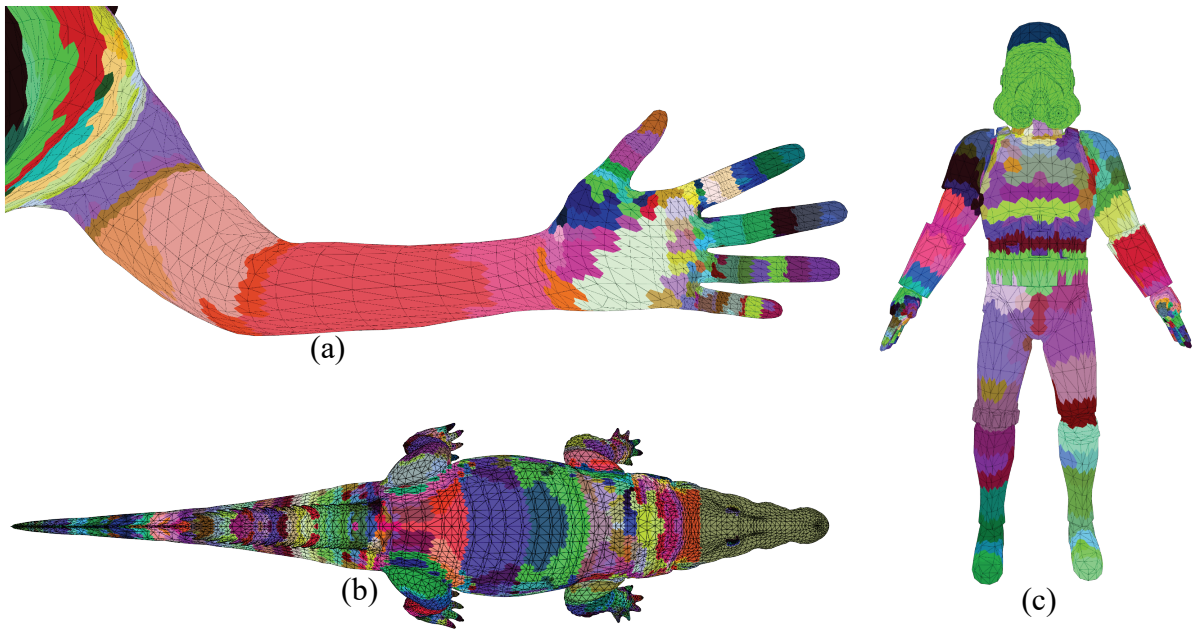


Fig. 7: visualization of unique index tuple regions

This coherence can be utilized by saving all unique tuple values in a set. Every vertex then receives a pointer (or rather another index) to that set entry containing the actual index tuple. The number of bits saved depends on the number of unique tuples. In the worst case, every vertex has its own unique tuple, increasing the needed storage by $V \cdot \log_2(V)$ with V being the

total number of vertices. Table 1 shows the compression ratio achieved for our test models. The ratio for an n-tuple of indices is given by:

$$f(S,n,V,N) = \frac{V \cdot n \cdot \lceil \log_2(N) \rceil}{V \cdot \lceil \log_2(S) \rceil + S \cdot n \cdot \lceil \log_2(N) \rceil}.$$

As can be seen, the compression ratio is between 2:1 and 2.5:1 for unsorted indices.

mesh	vertices	bones	unsorted set size	sorted set size	ratio unsorted	ratio sorted
*	V	N	S_1	S_2	$f(S_1,4,V,N)$	$f(S_2,4,V,N)$
croc	12800	109	1115	446	2.1:1	2.8:1
human	20340	53	420	191	2.5:1	2.9:1
boss	5828	53	387	188	2.3:1	2.7:1
archer	12424	54	358	178	2.5:1	2.9:1
dragon	22844	119	1330	599	2.2:1	2.6:1
cat	4969	58	545	230	1.9:1	2.6:1
trooper	5174	53	306	142	2.3:1	2.8:1
face	11371	112	1098	533	2:1	2.5:1
turtle	4346	24	163	85	2.3:1	2.7:1

Tab. 1: bits saved by introducing an index tuple LUT indirection

3.2 Further Set Size Reduction

It is possible to decrease the size of the index tuple set further by sorting the values inside a tuple. When swapping two index values, the corresponding weight values must be swapped as well. Sorting is possible because the order of the blending attributes within a vertex is usually arbitrary. For example, the two unique tuples $(14, 53, 0, 1)$ and $(53, 14, 0, 1)$ will get merged to one set entry $(0, 1, 14, 53)$. As can be seen in table 1, sorting the indices in a tuple improves the compression ratio to a value between 2.5:1 and 2.9:1. In section 4, we will instead sort the weights, making index sorting incompatible to the weight compression methods described in this thesis.

3.3 Shrinking the Lookup Table

While the bits needed per vertex usually is the dominant factor for the size of a mesh, it is also possible to reduce the size of the lookup table containing the unique index tuples. This gets relevant when a mesh has a lot of unique tuple sets or the table must fit inside a smaller memory

segment in the underlying hardware (e.g., constant memory of a GPU). Therefore, this section only outlines ideas that may get relevant in future work.

Because of the added indirection with the lookup table (*LUT)*, it is possible to make use of the different sizes of index tuples. Instead of saving all tuples in a single array and filling empty spots with invalid values, there can be an array for every tuple size. To differentiate which array should be selected, $n-2$ threshold values are saved, with n being the maximum number of tuple elements. If the set index surpasses such a threshold, a different array containing a different size of tuples is selected.

Another possible compression optimization is to convert the n index values to a single value using the method shown in the end of section 2.3.2. This is beneficial when the number of bones is not a power of 2 since gaps of unused bit states are closed.

4 Compressing the Weights

For every n-tuple of indices described in section 3, every vertex also has an n-tuple of weights. A weight usually is saved with a float16 or float32, requiring 2 or 4 bytes for every weight entry. In section 4.1 and 4.2 we will introduce techniques for reducing the sample space without changing the sample quality of the weights (lossless compression). In section 4.3 we will describe various methods for quantizing this reduced sample space (lossy compression).

4.1 Using the L1-Norm

The weight tuple is in an L1-Norm, meaning the sum of all weights is 1. This means, from a weight tuple $(w_1, w_2, w_3, \ldots, w_n)$ a single value w_i does not have to be saved and can be retrieved with:

$$w_i = 1 - \sum_{j=1, j \neq i}^{n} w_j.$$

Figure 8 visualizes this reduction: Every tuple got reduced by 1 dimension, also making it possible to visualize the 4-tuple in a 3D Projection (d). Additionally, the sample space of an n-dimensional weight tuple got changed from a (hyper-) cube to the simplex of the next lower space $(n-1)$. For example, a 4-tuple has valid values inside a tetrahedron (d), a 3-tuple inside a triangle (c) and so on.

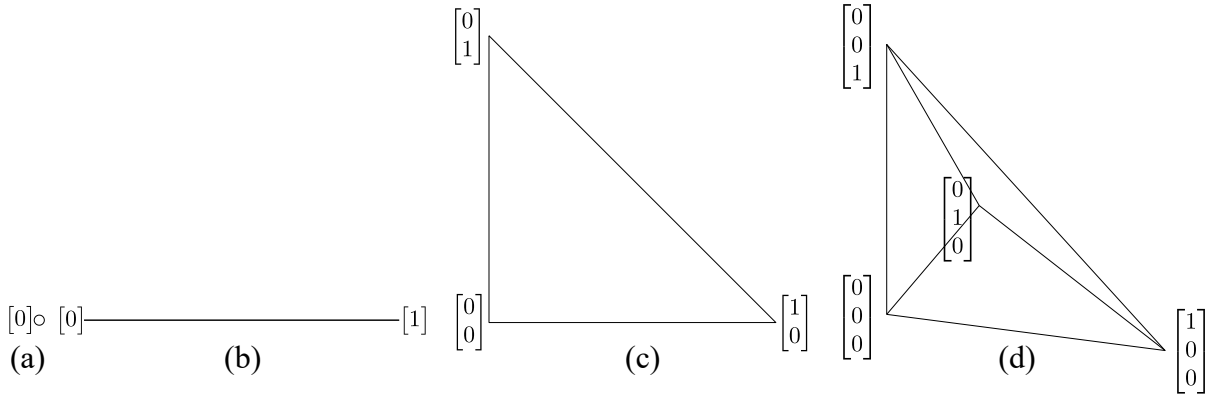


Fig. 8: weight sample spaces after using the L1-Norm

4.2 Sorting the Weights

By sorting the weight values inside a tuple, the sample size can be further reduced. A sorted weight tuple $(w_1, w_2, w_3, \ldots, w_n)$ fulfills the following directives:

$$w_1 \geq w_2 \geq \cdots \geq w_n; \quad \sum_{i=1}^{n} w_i = 1.$$

Since $w_j \geq w_i \; \forall \, j < i$, the case when $w_j = w_i \; \forall \, j < i$ and $w_k = 0 \; \forall \, i < k$ gives the maximum possible value for w_i:

$$i \cdot w_i = 1 \; \Rightarrow w_i = \frac{1}{i}.$$

For example, for a sorted tuple (w_1, w_2, w_3) the maximum valid value for w_3 is $\frac{1}{3}$. If it was higher $\left(\frac{1}{3} + \varepsilon\right)$, w_1 and w_2 must also be at least equal to $\frac{1}{3} + \varepsilon$. As $3 \cdot \left(\frac{1}{3} + \varepsilon\right) = 1 + 3\varepsilon > 1$ for $\varepsilon > 0$, higher values for w_3 violate the L1-Norm.

The minimum valid value for a weight stays at 0, except for the highest weight w_1. For w_1 to have its minimum value, all the other lower weights must have the same value $w_{1_{\min}} = w_2 = \ldots = w_n$:

$$\sum_{i=1}^{n} w_{1_{\min}} = 1 \quad \Rightarrow \quad n \cdot w_{1_{\min}} = 1 \quad \Rightarrow \quad w_{1_{\min}} = \frac{1}{n}.$$

When combining tuple sorting with the technique described in the previous section, it gets relevant to ask which value from a weight tuple is best left out and calculated from the other values to reduce the sample space the most. In the following, we will call the size of the sample space the *volume*, regardless of the dimension we are currently operating in. The volume of an m-dimensional simplex with the vertices $(v_1, v_2, \ldots, v_m, v_{m+1})$ with $v_i \in \mathbb{R}^{m \times 1}$ is defined by [Stein 1966]:

$$\left| \frac{1}{m!} \cdot \det \begin{pmatrix} v_1 & v_2 & \cdots & v_m & v_{m+1} \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \right|.$$

Every column i of the following matrix represents a weight n-tuple W_i. The values of every tuple are set so that the tuple element w_i has its maximum valid value $\frac{1}{i}$.

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n} \\ 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n} \\ 0 & 0 & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n} \\ 0 & 0 & 0 & \frac{1}{4} & \cdots & \frac{1}{n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{n} \end{pmatrix} \in \mathbb{R}^{n \times n}$$

By removing a row from that matrix, we receive the vertices of the simplex. In the following we will calculate the volumes of the 4 possible tetrahedrons created from a 4-tuple of weights:

$$\left| \frac{1}{3!} \cdot \det \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & \frac{1}{3} & \frac{1}{4} \\ 1 & 1 & 1 & 1 \end{pmatrix} \right| = \left| \frac{1}{3!} \cdot \det \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{4} \\ 1 & 1 & 1 & 1 \end{pmatrix} \right| =$$

$$\left| \frac{1}{3!} \cdot \det \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{4} \\ 1 & 1 & 1 & 1 \end{pmatrix} \right| = \left| \frac{1}{3!} \cdot \det \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{4} \\ 1 & 1 & 1 & 1 \end{pmatrix} \right| = \frac{1}{(n-1)!} \cdot \frac{1}{n!} = \frac{1}{144}.$$

As the volumes are always equal regardless of which row is removed, we can choose one. In the upcoming section about quantizing the volume, we always remove the largest weight w_1. This results in a simplex with easy-to-handle properties like one vertex at the origin of the coordinate system (0, 0, 0…) or an axis aligned edge. Figure 9 shows the resulting sample spaces for a 1-tuple (point) (a), 2-tuple (line) (b), 3-tuple (triangle) (c) and 4-tuple (tetrahedron) (d) with the respective vertex coordinates.

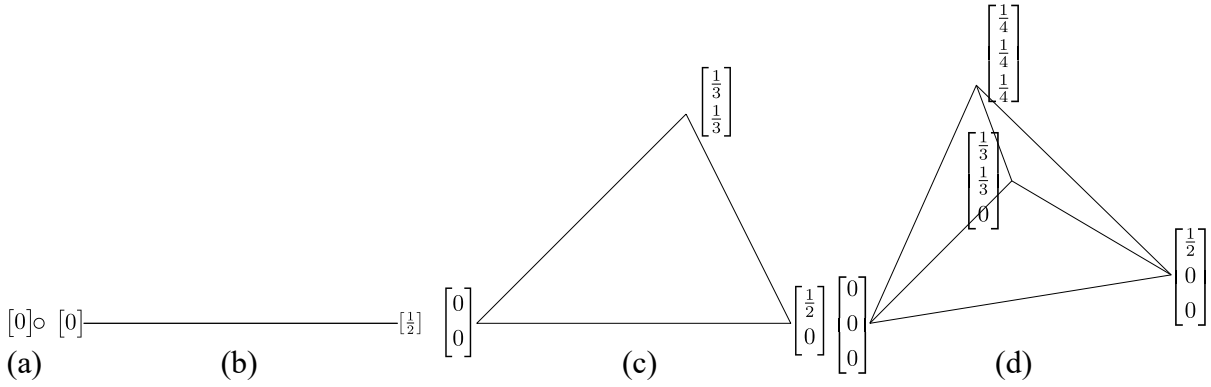


Fig. 9: weight sample spaces after sorting

To estimate the number of bits saved with this reduction, we can compare the volumes of the sample spaces. Coming from a volume created with the technique from 4.1, the number of bits saved with this new reduction for a weight n-tuple is

$$-\log_2\left(\frac{\frac{1}{(n-1)!}\cdot\frac{1}{n!}}{\frac{1}{(n-1)!}}\right) = -\log_2\left(\frac{1}{n!}\right) = \log_2(n!).$$

Table 2 compares the number of bits saved coming from an $(n-1)$-dimensional unit cube for a 1-, 2-, 3-, 4- and 8-tuple of weights. As can be seen, for an infinitely densely sampling, sorting saves up to ≈ 7.17 bits for a 4-tuple, as only 1 in 144 initial sample points actually are a valid weight tuple.

***n*-tuple**	**using the norm**		**norm and sorting**	
	simplex volume	bits saved	simplex volume	bits saved
n	$\frac{1}{(n-1)!}$	$\log_2((n-1)!)$	$\frac{1}{n!\cdot(n-1)!}$	$\log_2(n!\cdot(n-1)!)$
$n = 1$	0	0	0	0
$n = 2$	1	0	$\frac{1}{2}$	1
$n = 3$	$\frac{1}{2}$	1	$\frac{1}{12}$	≈ 3.58
$n = 4$	$\frac{1}{6}$	≈ 2.58	$\frac{1}{144}$	≈ 7.17
$n = 8$	$\frac{1}{5040}$	≈ 12.30	$\frac{1}{203212800}$	≈ 27.60

Tab. 2: bits saved with sample space reductions

4.3 Quantizing the Simplex

After reducing the sample volume of a weight n-tuple to an $n-1$ simplex, we need to sample the volume of that simplex. After decompressing an index set, the number of valid indices implies the number of weights. For example, with a decompressed index tuple $(14, 53, 8, -1)$ and -1 being an invalid index, we only need to sample the area of a triangle. Alternatively, without using the information from the index n-tuple about the number of non-zero weights, we would always have to sample the $n-1$ simplex. For our test meshes, about 51% of the vertices have a single index, 26% have 2, 12% have 3, and 11% have 4 indices. As we want our methods to directly work in a vertex shader, we sample every size of tuple with the same number of bits. By doing so, we accept that weight tuples of lower dimensionality are sampled more densely than tuples of higher dimensionality. Enabling different number of bits for different mesh regions, e.g., by using meshlets, could be topic of further research. In the following subsections, we describe 5 methods to quantize the simplex with methods 4 and 5 only working for up to a 4-tuple of weights and indices (tetrahedron).

To know when a lossless quantization is reached, we must evaluate how densely an original weight value in float16 or float32 is sampled. A float16 has a mantissa of 10 bit. Therefore, there are 2^{10} states for values between 0.5 and 1. For lower values the exponent is decremented, giving 2^{10} states for values between 0.25 and 0.5, 2^{10} states for values between 0.125 and 0.25 and so on. In other words, the closer the value gets to zero, the more densely the sampling becomes. As there is no apparent reason to sample a weight in such a non-uniform manner, we assume a lossless sampling to be 2^{10} states from 0.5 to 1 and 2^{10} states between 0 and 0.5. This gives us a total of $2^{11} = 2024$ states for a float16 and $2^{24} = 16777216$ states for a float32, which has a mantissa of 23 bits. This is similar to the assumptions made about float normal vector precision in [Meyer 2010]. Table 3 shows the bits required for an optimal simplex quantization to reach float16 quality together with the compression ratio. First, we take the initial size of the float tuple and reduce it to the actual sampling density of 11 bits per float for values between 0 and 1 (q). Next, we subtract the size of one value, as it can be calculated from the other tuple values (q'). Finally, we subtract the bits saved with the sorting technique introduced in the previous section. As can be seen, we reach a compression ratio of around 2.5:1. The same is also applied to a float32 scenario in table 4, where we reach a compression ratio of around 2:1.

n-tuple	float16	quantized q	save one q'	simplex s	ratio
n	$16 \cdot n$	$11 \cdot n$	$11 \cdot (n-1)$	$q' - \log_2(n! \cdot (n-1)!)$	$\frac{16 \cdot n}{s}$
$n = 1$	16	11	0	0	–
$n = 2$	32	22	11	10	3.2: 1
$n = 3$	48	33	22	≈ 18	$\approx 2.6: 1$
$n = 4$	64	44	33	≈ 26	$\approx 2.5: 1$
$n = 8$	128	88	77	≈ 49	$\approx 2.6: 1$

Tab. 3: bits required to reach f16 quality with an optimal quantization of the simplex

n-tuple	float32	quantized q	save one q'	simplex s	ratio
n	$32 \cdot n$	$24 \cdot n$	$24 \cdot (n-1)$	$q' - \log_2(n! \cdot (n-1)!)$	$\frac{32 \cdot n}{s}$
$n = 1$	32	24	0	0	–
$n = 2$	64	48	24	23	$\approx 2.8: 1$
$n = 3$	96	72	48	≈ 44	$\approx 2.2: 1$
$n = 4$	128	96	72	≈ 65	$\approx 2.0: 1$
$n = 8$	256	192	168	≈ 140	$\approx 1.8: 1$

Tab. 4: bits required to reach f32 quality with an optimal quantization of the simplex

4.3.1 Lloyd Cluster Quantization

A common method for finding good sample points for a quantization is to use the k-means clustering algorithm, first described by [Lloyd 1982]. This algorithm takes the uncompressed data and determines a given number of cluster centroids from it. The number of centroids 2^B is given by the desired number of bits B per vertex. We then save the centroids in a lookup table LUT and replace the weight tuple of every vertex with the index to the closest centroid entry. To use the information about the number of weights unequal to 0 (valid weights) given by the index tuple, we perform the cluster analysis for all different numbers of valid weights. This implicates that there will be a LUT for 2-, 3- up until n-tuples of valid weights.

Figure 10 visualizes a cluster quantization for a mesh with 2 bits or 4 centroids per tuple category. As can be seen, points “snap” to its closest centroid within its dimension (green to red). The 12 sample points are given by 4 Centroids per dimension multiplied by the 3 categories of tuples (2, 3, and 4).

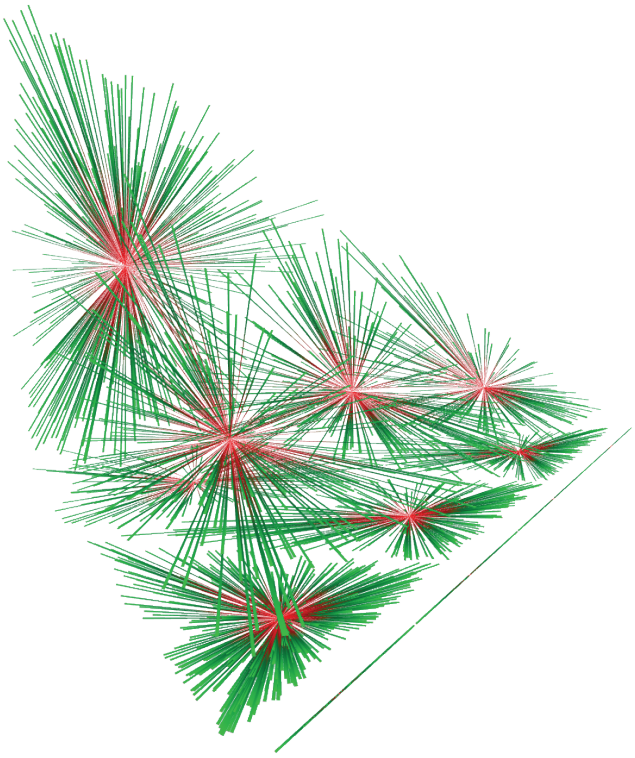
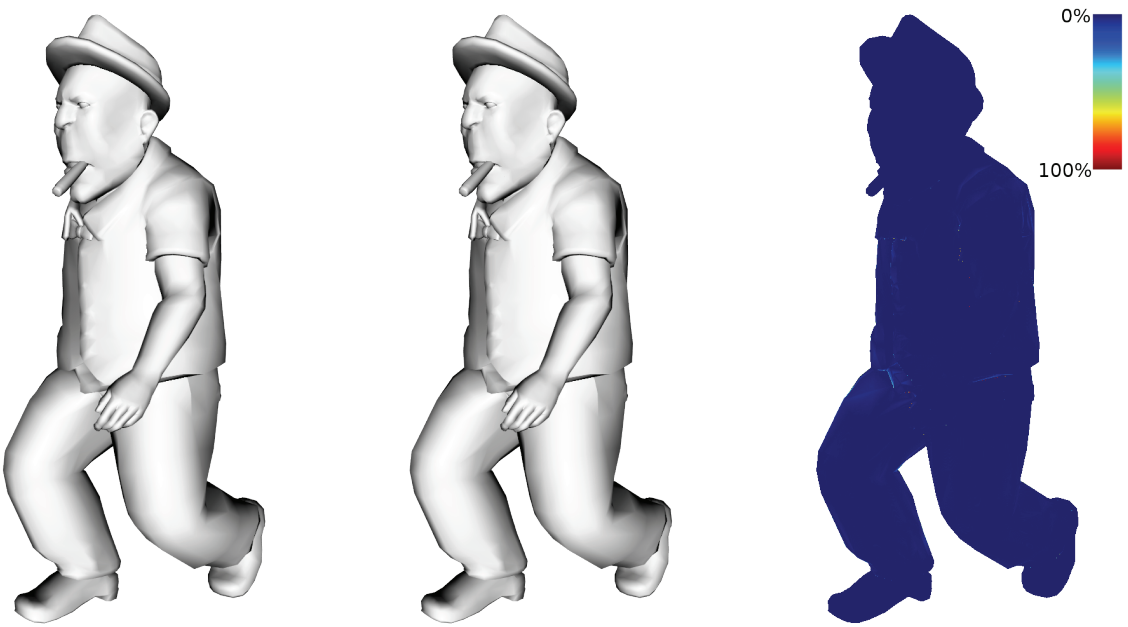


Fig. 10: weight clusters example

For n-tuples of weights we must save $n-1$ LUTs with 2^B elements. Therefore, cluster quantization is only reasonable to use with smaller number of bits B per vertex. In Figure 11 (a) the scene is rendered without compression. In (b) we use a weight clustering compression with 5 bits per vertex. In (c) we compare both images to each other pixel by pixel. Dark blue represents no difference in pixel intensity, while red color represents a fully contrary intensity. More information about this error visualization can be found in section 5.2. As can be seen, even with a low number of bits per vertex, clustering can result in near unrecognizable visual error for some meshes.



(a) (b) (c)

Fig. 11: mesh rendered with and without Lloyd Cluster weight quantization

With the number of vertices $|V| = 5828$, the number of bits per vertex $B = 5$, weight tuples of $n = 4$ dimensions and the number of bits required to save a single weight value $|\text{float32}| = 32$ bit, the compression ratio of the example is

$$\frac{|V| \cdot n \cdot |\text{float32}|}{|V| \cdot B + 2^B \cdot |\text{float32}| \cdot \sum_{i=1}^{n-1} i} = \frac{745984 \text{ bit}}{35284 \text{ bit}} \approx 21 : 1.$$

4.3.2 Tetrahedron Subdivision

To exceed the bit per vertex limit from previous section, a method for encoding sample points without involving a LUT is required. By splitting every side in half, an m-dimensional simplex can be subdivided into 2^m smaller simplexes of equal volume. These smaller simplexes can be subdivided again. After k subdivision steps the initial simplex is split into $2^{m \cdot k}$ parts. By enumerating the subdivision elements and concatenating these numbers, we create an encoding to address a sub-simplex. Figure 12 demonstrates that for a triangle. In (a) an example for an enumeration is given. Every sub-simplex gets a number (0 – 3 in bit representation). In (b) we recursively subdivide starting at the bottom right (00), continuing with the top (10) and finishing with the center (11) subdivision, resulting in the blue sub-triangle.

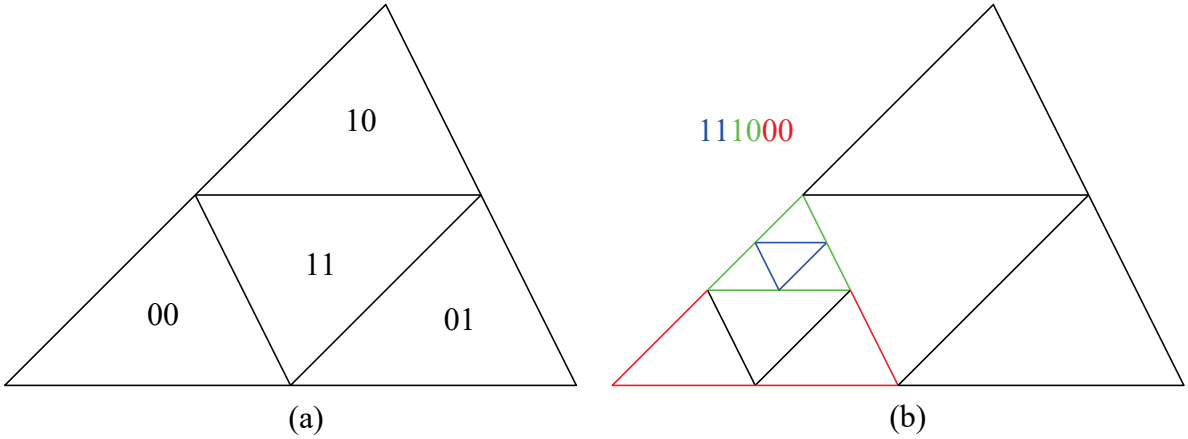


Fig. 12: subdivision enumerating and encoding example

Next, the sample point is defined by the *centroid* of the sub-simplex. The centroid is given by the average position of the vertices of the simplex. To reduce the maximum error, given by the distance of the sample point to the vertices, the *circumcenter* would be a more optimal choice. The circumcenter is the center of the (hyper-) sphere that passes through all the vertices of the simplex. Due to a signification algorithm optimization which will be addressed later, the less optimal centroid is used instead. Figure 13 shows that the circumcenter (red) and the centroid (green) of the triangle are quite similar.

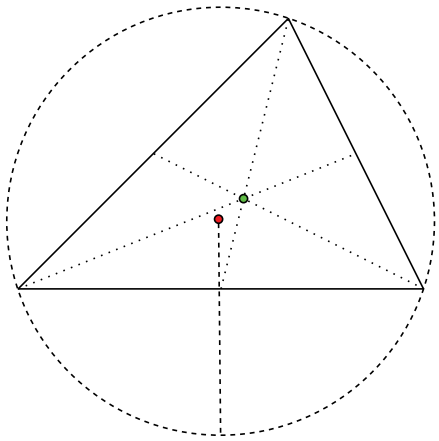


Fig. 13: comparison between the circumcenter and the centroid

A simplex can be described by a matrix with the column vectors representing the vertices of the simplex. Every subdivision step can also be described by a matrix. Every row contains the information about which of the original vertices are used by the new vertices by how much (fully or half). Using this definition, a subdivision step is a matrix-matrix multiplication. With i, j and k being indices describing a subdivision created from the previously introduced

encoding and the initial triangle T, the sub-triangle T' can be retrieved with the following algorithm:

$$S = (s_0, s_1, s_2, s_3) = \left(\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix}, \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 1 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{bmatrix}, \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} \right); T = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{3} \\ 0 & 0 & \frac{1}{3} \\ 0 & 0 & 0 \end{bmatrix};$$

$$T' = T \cdot S_i \cdot S_j \cdot S_k.$$

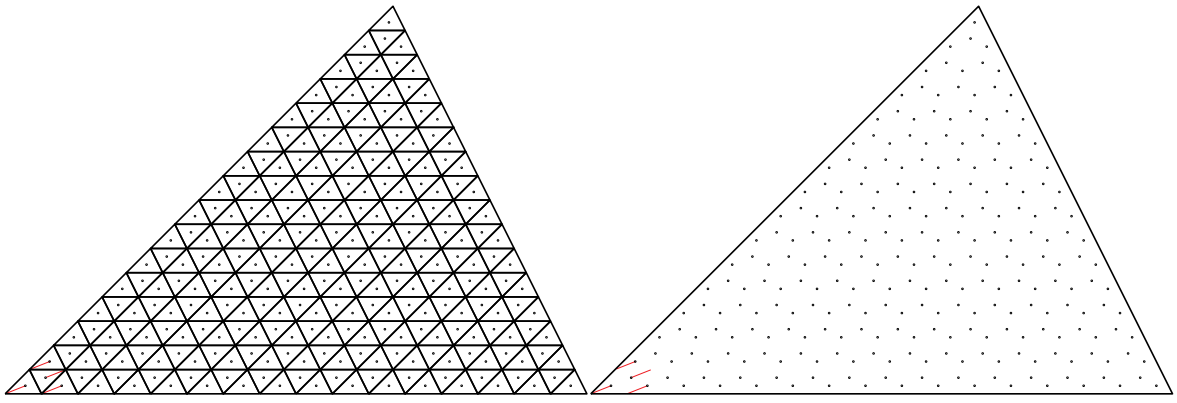
To retrieve the sample point centroid, we take the average of the vertices of the new triangle:

$$C = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \quad P_{ijk} = T' \cdot C \; = T \cdot S_i \cdot S_j \cdot S_k \cdot C.$$

As matrix multiplications are associative, we can perform the already mentioned optimization by replacing all matrix-matrix multiplications with matrix-vector multiplications:

$$P_{ijk} = T \cdot (S_i \cdot (S_j \cdot (S_k \cdot C))).$$

Figure 14 visualizes the sample point distribution for $m = 2$ and $B = 8$ (8 bit $= 256$ Points). In (a) the resulting sub-triangles are shown together with the sample points. In (b) the edges of the sub-triangles are removed. In both images red lines give an example of where quantization of a point would result in the maximum quantization error.



(a) (b)

Fig. 14: subdivision sample point distribution

As can be seen, the sample points are not arranged in a way that minimizes the quantization error: sampling holes emerge from the honeycomb-like pattern. Additionally, the decompression runtime grows with the number of subdivisions and therefore with the number of bits per vertex as more and more matrix-vector multiplications must be computed. Suffering from these issues, this method, while being easy to understand and implement, is unsuited for our real-time application.

4.3.3 Cuboid Quantization

This method quantizes the axis-aligned bounding-box *AABB* of the tetrahedron, resulting in a cuboid as a sample space. The way quantization of a (hyper-)cuboid works is described in section 2.3.2. The axis lengths $(l_1, l_2, \dots, l_m)$ of the AABB of an m-dimensional weight simplex are given by:

$$l_i = \frac{1}{i+1}.$$

The following table 5 compares the volumes of the simplexes and their AABB volume and gives an estimation of the bits lost by sampling the larger volume. While not being optimal, sampling the AABB can still be a valid strategy for a low number of tuple elements as only a few bits of invalid sample points are unnecessarily included.

***n*-tuple**	**simplex volume**	**bounding box volume**	**bits lost**
n	$\frac{1}{n! \cdot (n-1)!}$	$\frac{1}{n!}$	$\log_2((n-1)!)$
$n = 1$	0	0	0
$n = 2$	$\frac{1}{2}$	$\frac{1}{2}$	0
$n = 3$	$\frac{1}{12}$	$\frac{1}{6}$	1
$n = 4$	$\frac{1}{144}$	$\frac{1}{24}$	≈ 2.58
$n = 8$	$\frac{1}{203212800}$	$\frac{1}{40320}$	≈ 12.30
$n = 16$	$\frac{1}{16 \cdot (15!)^2}$	$\frac{1}{16!}$	≈ 40.25

Tab. 5: bits lost by sampling the simplex AABB

4.3.4 Sheared Tetrahedron Chopping

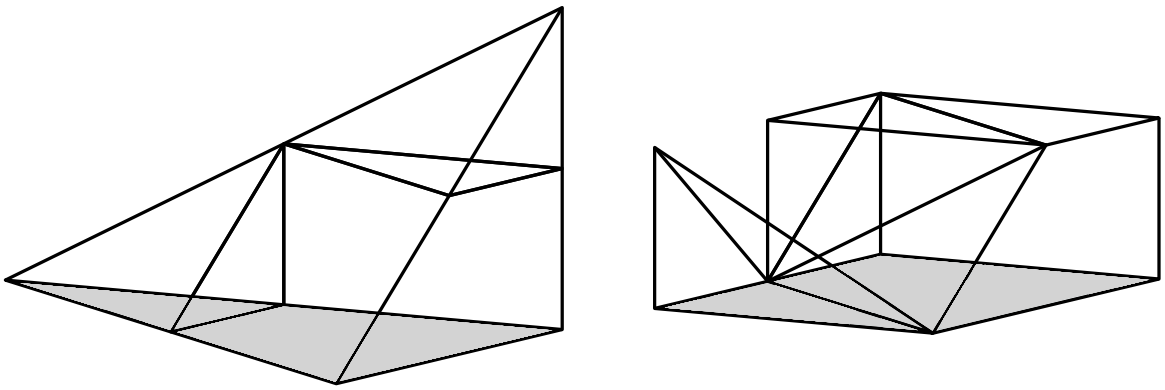
To better fit in a cuboid, and therefore to reduce the number of unusable sample points from the previous method, we chop the simplex into multiple pieces and transform them. This is done for the 3- and 4-tuple only, therefore this method only works for weight n-tuples $n \leq 4$. The first transformation shears the tetrahedron T (or triangle if the last row gets removed) with the matrix S:

$$S = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}; T = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & \frac{1}{3} & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{4} \end{bmatrix}; T' = S \cdot T = \begin{bmatrix} 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{4} \end{bmatrix}.$$

Alternatively, this transformation can be interpreted as a delta encoding: we only save the difference between a weight and the next weight. Next, we split the tetrahedron into 3 (2 for the triangle) parts. A weight vertex $v = (w_2, w_3, w_4)$ is assigned to one of the parts A, B and C following the directives:

$$v \in \begin{cases} B, & \text{if } w_2 > \frac{1}{4} \\ C, & \text{if } w_4 > \frac{1}{8} \\ A, & \text{Otherwise} \end{cases}.$$

Vertices of part B and C are then rotated and mirrored. Figure 15 (a) visualizes the sheared tetrahedron and the 3 parts. In (b) the parts are shown after the transformation. The grey shaded faces resemble the same transformation for a triangle. As can be seen, the sample-triangle (grey) has now the shape of a rectangle and the sample-tetrahedron now better fits the shape of a cuboid.



(a) (b)

Fig. 15: creation of a cuboid shape by chopping the tetrahedron into parts

Lastly the cuboid and rectangle are quantized like described in section 2.3.2. Additionally, sample points lying on a partitioning plane must be prevented. This is easiest done by adjusting the rounding strategy described in the quantization chapter to not allow an odd number of points for the w_3 axis.

Table 6 shows that, with this method, less than a bit is lost when sampling 4-tuples, while no bits are lost for 3-tuples.

n-tuple	simplex volume	chopped box volume	bits lost
$n = 3$	$\frac{1}{12}$	$\frac{1}{12}$	0
$n = 4$	$\frac{1}{144}$	$\frac{1}{96}$	≈ 0.58

Tab. 6: bits lost by sampling the chopped box

4.3.5 Sheared Tetrahedron Quantization

With this method we enumerate sample points lying inside the simplex in a quadratic grid. To simplify the enumeration, we again shear the simplex like described in the previous section. Figure 16 shows an example of the point distribution for the triangle. We describe the grid with the number of sample points $n = 6$ in the bottom row. As the bottom edge of the triangle is already sampled by the next lower simplex (line), we offset the grid by half the distance between two neighboring points, which is also equal the maximum quantization error E. The offset can be calculated for every n by solving the equation:

$$\frac{1}{2} - \frac{3}{2}E = 2 \cdot (n-1) \cdot E \;\Rightarrow\; E = \frac{1}{4n-1}.$$

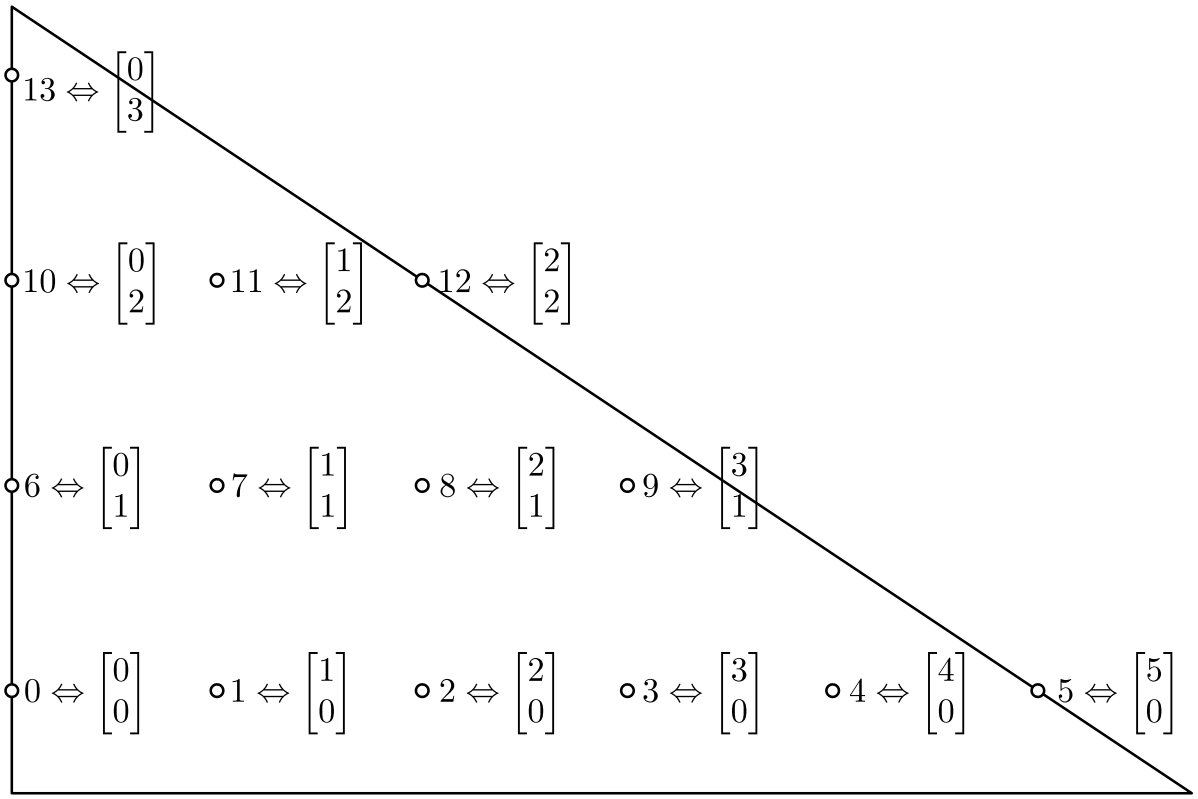


Fig. 16: sample point enumeration for the triangle with 1D and 2D indices

Every point in the triangle (or tetrahedron) has a 1D and a 2D (or 3D) index. The 2D index corresponds to the 2D position of the point and is generated similarly to the cuboid quantization from section 2.3.2. The 1D index is the value saved as the vertex attribute. When decompressing, we must separate that value into the 2D index. For this we need a function *baseIdx3* that gives the 1D index $\mathbb{I}$ of the first point of every row, given the row index I_3. We call it I_3 as it is directly connected to the weight tuple value w_3. Given the number of points in the bottom row $n = 6$, the baseIdx3 of a row is given by the sum of the number of points of the rows below. As the number of points in a row decreases by the gradient of the hypotenuse of the triangle $(-3/2)$, it is given that

$$\text{baseIdx3}(I_3, n) = \sum_{i=0}^{I_3-1} \left\lfloor n - \frac{3}{2}i \right\rfloor = \left\lfloor \frac{-3{I_3}^2 + 2I_3 + 1}{4} + I_3 \cdot n \right\rfloor.$$

Figure 17 plots this function for $n = 6$. Comparing this plot with figure 16 shows that for every row ($0, 1, 2$ and 3), baseIdx3 returns the 1D index of the first element of the row.

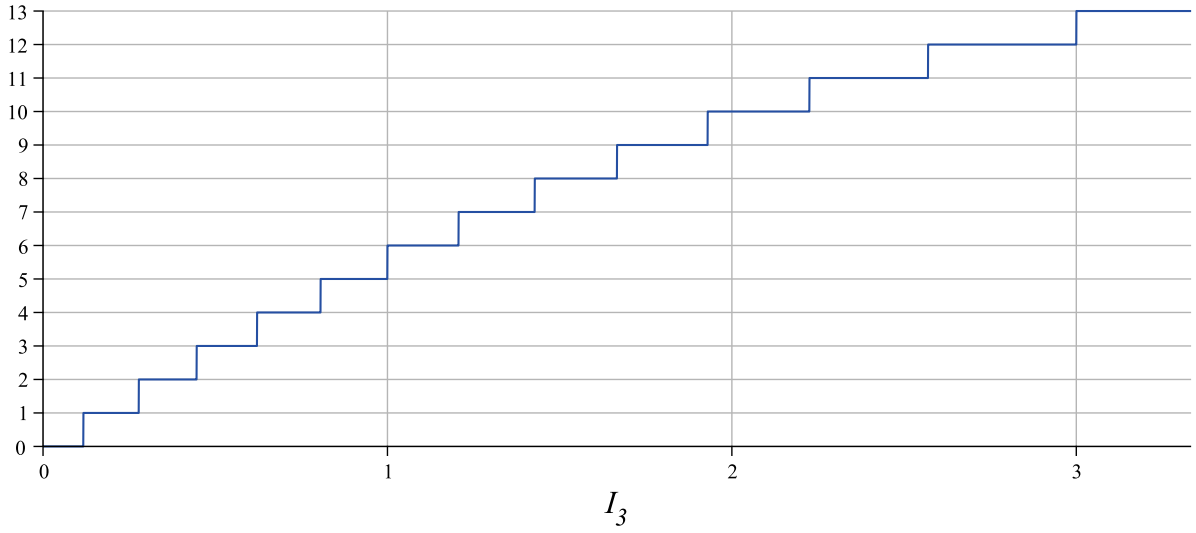


Fig. 17: baseIdx3 plot

The GLSL code in figure 18 shows the decompression algorithm. We receive the compressed weight tuple $\mathbb{I}$ and the number of bones from the index tuple as arguments. For our example case `nBones=3` we first calculate I_3(`i3`) from $\mathbb{I}$, using the inverse function of baseIdx3 (line 21). Next, we calculate the weight value `c` (w_3), using the maximum error value given by the uniform variable `in_maxErr3`. Subsequently, we call the decompression function of the next lower weight. As an argument, we must reduce $\mathbb{I}$ by subtracting the baseIdx3 of I_3 so that it represents the location inside the current row. After unpacking the weight values from the indices, we add the offset to the next lower sample space (line 4, 5 and 6 before the break). Lastly, we shear the weights back and calculate the first weight from the other values (line 10-13). The decompression of a 4-tuple works analogously to a 3-tuple, additionally requiring a baseIdx4 function and its inversion.

```
1  vec4 decompressWeights (uint I, uint nBones){
2          vec3 bcd = vec3(0);
3          switch (nBones) {                                  //+ offset to next lower sample space
4          case 2u: bcd.x = decomp2(I, in_maxErr2)              + in_maxErr2; break;
5          case 3u: bcd.xy = decomp3(I, in_nBot3, in_maxErr3)  + vec2(0, in_maxErr3); break;
6          case 4u: bcd.xyz = decomp4(I, in_nBot4, in_maxErr4) + vec3(0, 0, in_maxErr4); break;
7          default: return vec4(1, 0, 0, 0);
8          }
9          //shear alias delta code
10         bcd.y += bcd.z;
11         bcd.x += bcd.y;
12         float a = 1. - dot(bcd, vec3(1.));
13         return vec4(a, bcd);
14 }
```

```
15  float decomp2(uint I, float error){
16          uint i2 = I;
17          float b = float(i2) * 2. * error;
18          return b;
19  }
20  vec2 decomp3(uint I, uint n, float error){
21          uint i3 = baseIdx3inv(I, n);
22          float c = float(i3) * 2. * error;
23          return vec2(decomp2(I - baseIdx3(i3, n), error), c);
24  }
25  vec3 decomp4(uint I, uint n, float error){
26          uint i4 = baseIdx4inv(I, n);
27          float d = float(i4) * 2. * error;
28          return vec3(decomp3(I - baseIdx4(i4, n), n - 2u*i4, error), d);
29  }
```

Fig. 18: Sheared Tetrahedron Quantization decompression GLSL code

Table 7 shows that, for common numbers of bits per vertex B, an n exists so that most of the states of these bits represent a sample point. This ratio is calculated using the baseIdx functions with the number of indices in direction of I_3 and I_4 as an argument:

$$\frac{\text{baseIdx3}\left(\left\lfloor\frac{2}{3}\cdot n+\frac{1}{3}\right\rfloor,n\right)}{2^B} \text{ and } \frac{\text{baseIdx4}\left(\left\lfloor\frac{n}{2}\right\rfloor,n\right)}{2^B}.$$

B	n for triangle	points in triangle	states used	n for tetrahedron	points in tetrahedron	states used
4	6	14	87.5%	5	15	93.75%
8	27	252	≈ 98.44%	15	249	≈ 97.27%
16	442	65269	≈ 99.59%	104	65231	≈ 99.53%
32	113511	4294953544	≈ 100%	4258	4293419700	≈ 100%

Tab. 7: usable bit states for Sheared Tetrahedron Quantization

As 50% of unused states would mean that 1 bit is wasted, this method is superior to the static 0.58 bits lost by the chopping technique introduced before. As it is hard to find the baseIdx formula and especially its inversion in higher dimensions, this method is limited to 4-tuples of weights. Expanding this limitation might become subject of future work.

5 Evaluation

In this section we evaluate the presented compression techniques by testing and describing different evaluation metrics.

5.1 Geometric Error

This error metric shows by how much the usage of a lossy weight compression modifies the animated geometry of the mesh. For every keyframe of the animations of a mesh, we compare the animated position of every compressed vertex to the position of a vertex animated with uncompressed weights. To make the results more comparable between different mesh sizes, the distance values are normalized with the AABB diagonal length of the mesh. A use case for this error metric could look like the following: The user defines a maximum geometric error threshold that is not allowed to be exceeded. An algorithm can then search for the optimal compression ratio for a mesh and its animations which fulfills that requirement. A scenario when the geometric error is low compared to a high compression ratio could look like the following: Vertices are influenced by 2 bones. These 2 bones have a very similar transformation (e.g., 2 spine bones). When playing the animation, the exact weight values do not matter, as the vertex gets transformed to the correct position anyway. This can also be applied to our test meshes: In figure 19, we plot the maximum Geometric Error for our test meshes for a different number of bits per vertex using the Sheared Tetrahedron Quantization method. We want the compression to have a maximum error of 0.5% of the meshes AABB diagonal. As can be seen, some meshes have a lower error curve and are therefore more resistant to compression artifacts. For example, “croc_swim” reaches the desired quality with only around 5 bits. Contrary to this, the “dragon_idle” mesh requires at least 12 bits to reach our threshold. We visualize the cause of this in figure 20: the heat color indicates the Geometric Error. We can see that the complex folding and unfolding of the dragon’s wings requires more precision.

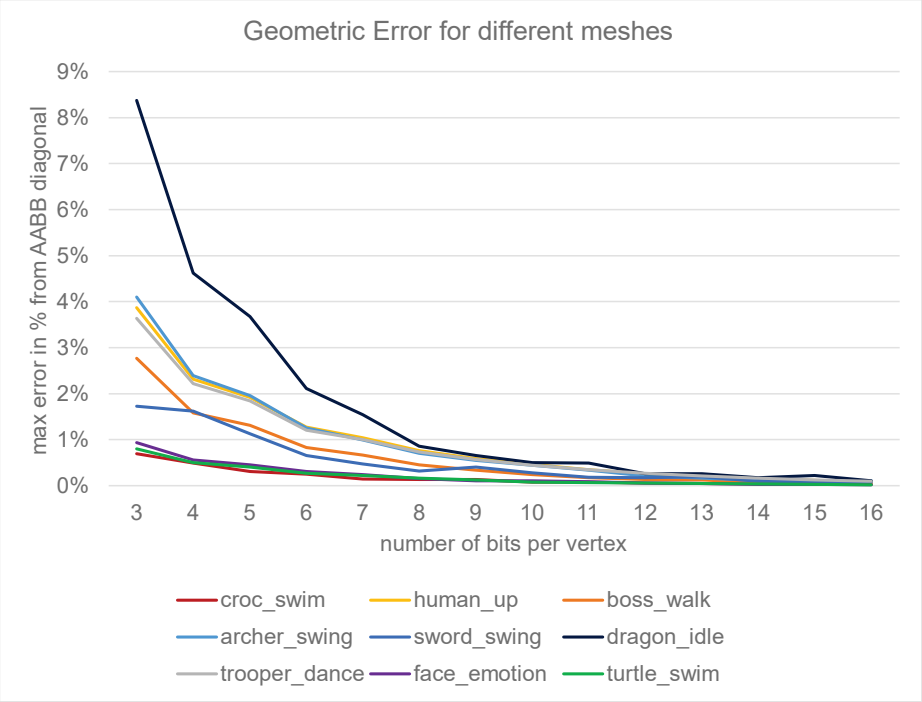


Fig. 19: Geometric Error plot for different meshes

Fig. 20: animation with high Geometric Error

In figure 21, we visualize the average geometric error for the same mesh but compressed with the different compression methods. In (a) we plot the error against the bits per vertex, while in (b) we plot the error against the actual compression ratio. As expected, the Lloyd Cluster method performs best for low bits per vertex. Unfortunately, the method requires a LUT. Therefore, the actual compression ratio is worse than for the other methods when using the same number of bits per vertex. The “sweet spot” to switch from Lloyd Cluster to the Sheared Tetrahedron Quantization method seems to be at around 5 bits or a ratio of 26:1.

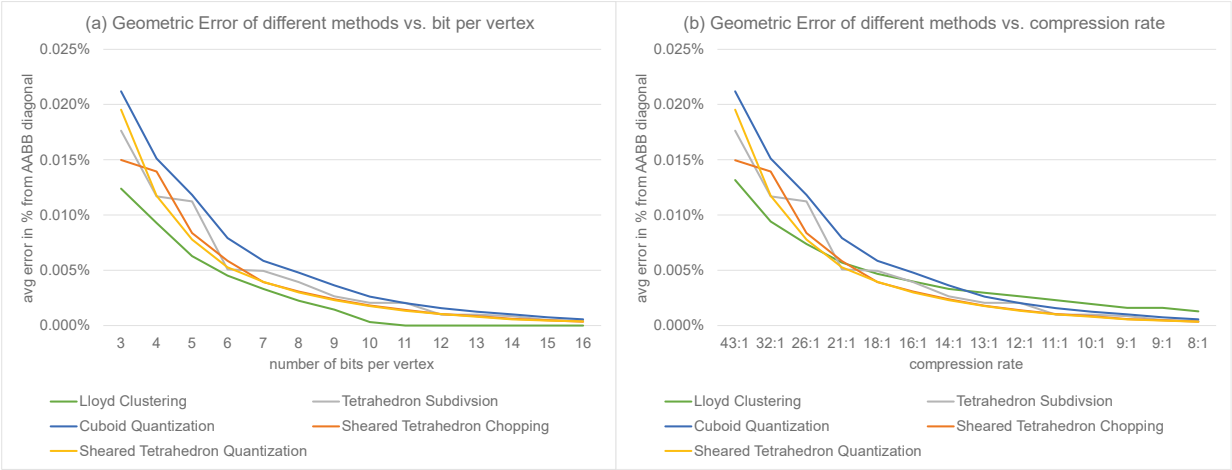


Fig. 21: Geometric Error plot for different methods

5.2 Visual Error

With this metric, we render each frame twice: once with and once without compressed weights. Next, the resulting images are compared pixel-by-pixel. A minor or no difference between the two images indicate a sufficient compression ratio: further increasing the weight precision does not increase the quality of the rendered image. Figures 22 and 23 show the visual error for weight tuples compressed into 4, 8, 12, and 16 bits for two of our test meshes.

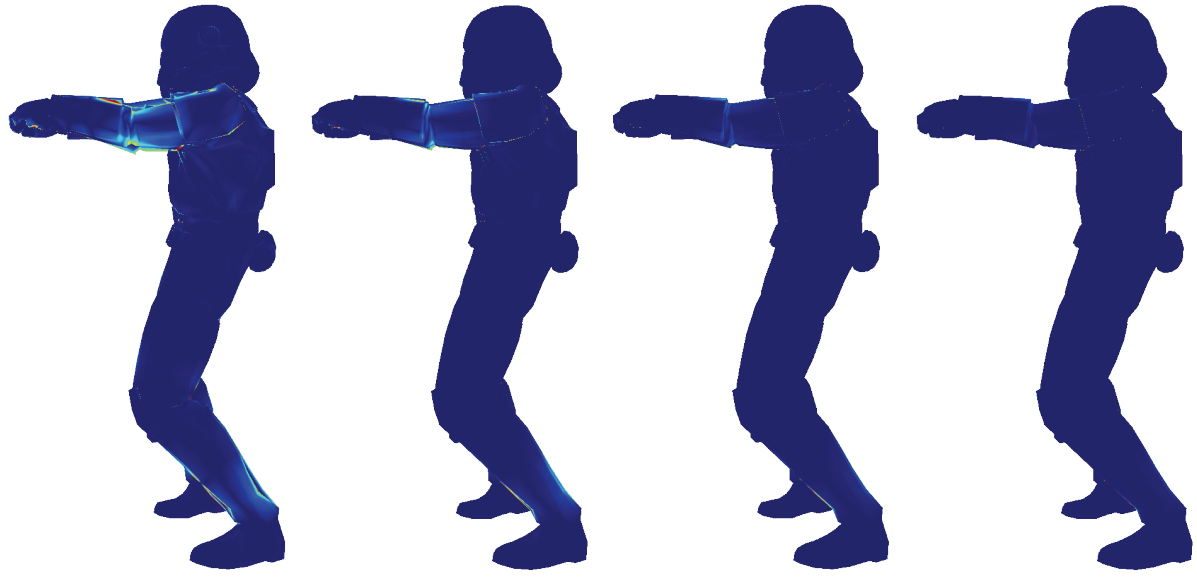


Fig. 22: Visual Error of STQ method for different bits per vertex example 1

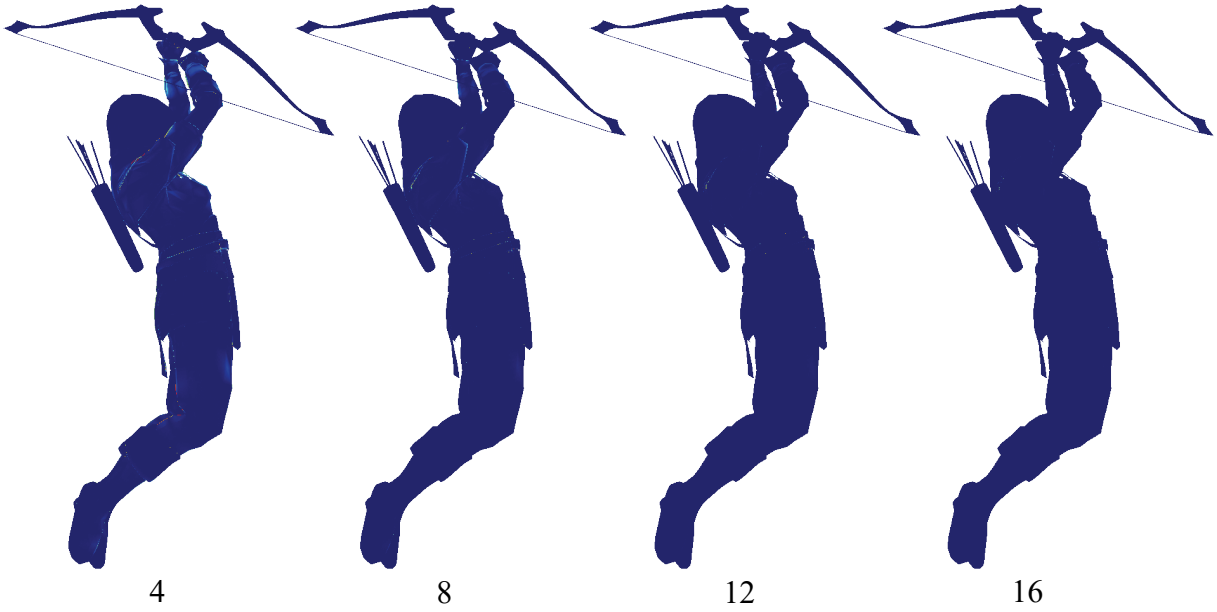


Fig. 23: Visual Error of STQ method for different bits per vertex example 2

5.3 Benchmarks

In this section we present benchmarks of our methods. We perform all measurements with an OpenGL compute shader on a NVIDIA GTX 1060 graphics card. In our benchmark, we measure the decompression time of our methods and compare it with the time required to load the uncompressed data. We exclude the Lloyd Cluster method from this test and evaluate it differently for reasons described later in this section. In our test scenario we want to decompress $2^{24} \approx 16M$ weight 4-tuples compressed with 32 bits for every tuple. First, we measure the time it takes for the GPU to just copy $4 \cdot 4 \cdot 2^{24}$ bytes. This simulates the scenario of a vertex shader loading a 4-tuple of uncompressed float32 weight data. While a vertex shader usually does not store the weight data again and instead immediately uses it to compute the animated position of the vertex, we include the time to store the 4-tuple in all our measurements. Next, we measure the compute time for each of our methods. This measurement includes the time required to read the compressed data ($32\ \text{bit} = 4\ \text{byte}$), to decompress it, and to store the result. To simulate the additional information about the number of bones from the index tuple, we repeat this for 1, 2, 3 and 4 bones. We perform all measurements multiple times and take the median of the distribution.

Figure 24 visualizes the results. Each colored bar represents a compression technique. With a single exception, the methods perform similarly, laying slightly above the time it takes to read

the 4 bytes of compressed data from memory and to store the uncompressed 4-tuple (dotted line). For most testcases, the actual decompression computation time is overlayed by memory latency and bandwidth limitations. This technique is called *latency-hiding*: while decompressing a weight, the GPU-thread can store data from the previous task or load data for the next task in parallel [Volkov 2016]. Roughly speaking, if the computation is faster than the memory latency, the values for loading and computing do not add up and only the loading time is observed. The high value for the 2-weight Tetrahedron Subdivision decompression can be explained with the already mentioned expensive loop in the decompression algorithm: with compressed data of 32 bits the GPU must compute 33, 17, and 11 matrix-vector multiplications for a 2-, 3-, and 4-tuple of weights. When comparing the values with the time it takes to load an uncompressed 4-tuple (dashed line), we observe that for most of our testcases the usage of a compression technique (loading compressed data and decompressing it) is faster than loading the original data. Therefore, our compression does not only reduce memory usage, but also can speed up rendering if the applications performance is bandwidth limited.

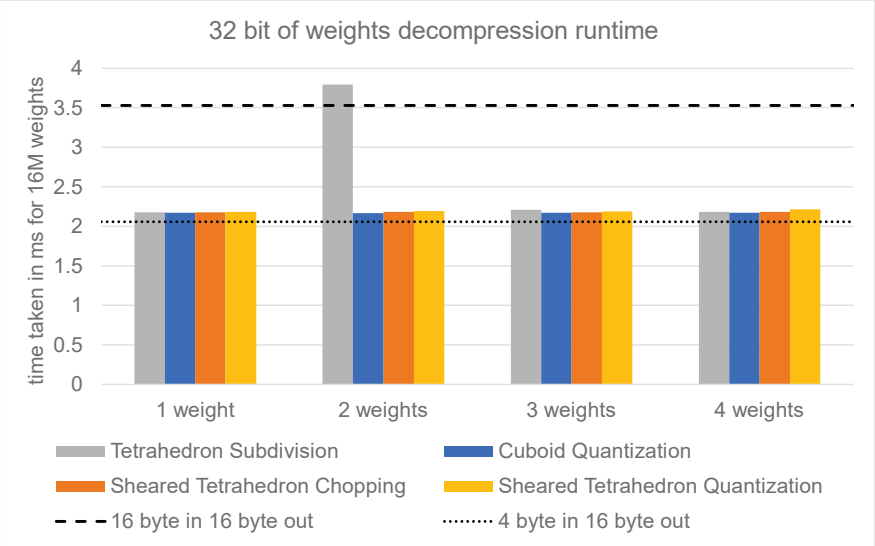


Fig. 24: benchmark results comparing our methods and no compression

We excluded the Lloyd Clustering method as it is unsuited for such high values per vertex as 32 bits: a mesh would require over 12 billion vertices for the clustering to find 2^{32} centroids for 2, 3, and 4-tuples of weights. Alternatively, as the methods performance is directly dependent on the memory access speed to the centroid LUT, we iterate over different number of bits per vertex. As can be seen in figure 25, the compute time stays slightly above the time to load the LUT index of 4 bytes for LUT sizes smaller than $2^{12} = 4096$ entries. For our testcase, the LUT stores 3-tuples of float32s (largest weight gets calculated from the other

values) which corresponds to $2^{12} \cdot 3 \cdot 4B = 48\,KiB$ of data. Our GPU has a global memory read (L1) cache of the same size (48 KiB) [NVIDIA 2021]. Therefore, we speculate that for larger LUT sizes, the graphics card runs into a growing number of cache misses and thus requires longer compute times. As this depends a lot on the hardware used, we suggest that to find good LUT sizes for vertex attribute compression applications should be a topic of future research. This also applies to the index set compression technique, which works equivalently.

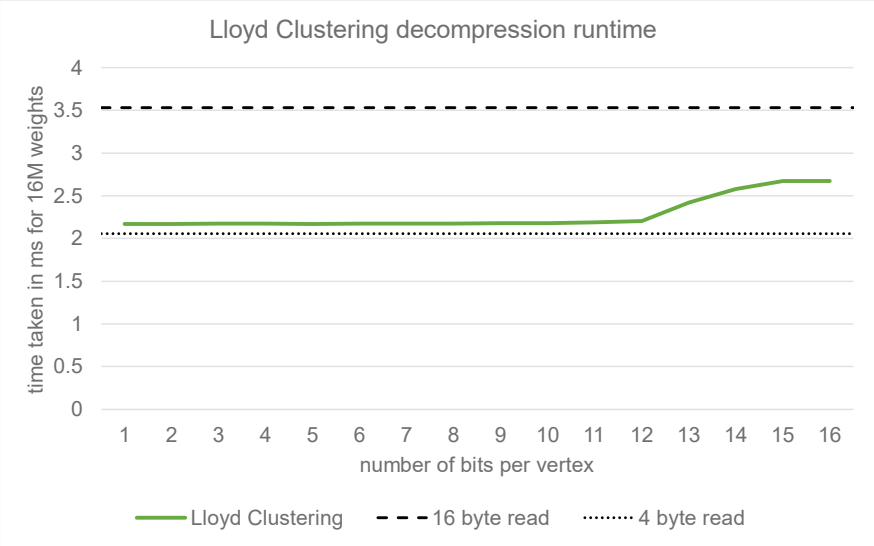


Fig. 25: Lloyd Cluster runtime comparison for different LUT sizes

6 Conclusion and Outlook

In this thesis, we presented techniques for compressing vertex-blending attributes used for skeleton-based animation. For the bone indices, we made use of patches of equal index tuples. Every unique index tuple is saved in a LUT with every vertex receiving a reference to its LUT entry. Our methods accomplish a compression ratio between 2:1 and 2.9:1, depending on the implementation and tested mesh.

For the weights, we first reduced the sample space to a simplex by using the fact that the weights are in an L1-Norm. We further shrunk that simplex by sorting the weight values inside a tuple. With an optimal quantization of that simplex, this sampling space reduction can achieve a lossless compression ratio of around 2.5:1 compared to a float16 representation. Subsequently, we presented several methods for quantizing the reduced space and discussed advantages and disadvantages of these methods. We described 2 methods that can sample the simplex with less than a bit lost for up to a 4-tuple of weights. While a maximum of 4 bone influences per vertex has been industry standard for many years for real-time rendering applications, recent developments increase this limit to 8 or even an unlimited number of influences. Therefore, to find methods for efficient quantization of the volume of an n-dimensional simplex should be topic of further research.

In our evaluation, we showed that for some meshes and applications a lossy quantization with a compression ratio of up to 20:1 can produce images with near unrecognizable error. We suggested to tweak for the best compression ratio for every mesh and its animations separately, using a geometric or visual error metric. Finally, we showed that our compression methods can speed up rendering for bandwidth limited applications. As decompression is very cheap, the saving is directly dependent on the reduced loading time of the data.

References

[Crytek 2015] Crytek: Biped Rigging Documentation, https://docs.cryengine.com/display/SDKDOC2/Biped+Rigging#BipedRigging-Skinning (Accessed: 01.01.2021).

[Frechette 2017] Frechette, N.: Simple and Powerful Animation Compression, GDC 2017, https://www.gdcvault.com/play/1024009/Simple-and-Powerful-Animation (Accessed: 25.01.2021).

[Frey 2011] Frey, I.; Herzeg, I.: Spherical Skinning with Dual Quaternions and QTangents, ACM SIGGRAPH 2011 Talks.

[Granlund 1994] Granlund, T.; Montgomery, P. L.: Division by Invariant Integers Using Multiplication, ACM SIGPLAN, 1994.

[Kavan 2008] Kavan, L.; Collins, S.; Zara, J.; O'Sullivan C.: Geometric Skinning with Approximate Dual Quaternion Blending, ACM Transactions on Graphics, Vol. 27, Nr. 4, Article 105, 2008.

[Kenwright 2012] Kenwright, B.: A Beginners guide to Dual-Quaternions - What They Are, How They Work, and How to Use Them for 3D Character Hierarchies, Winter School of Computer Graphics, Vol. 2, 2012.

[Khronos 2016] Khronos Group Inc: glTF Readme, https://github.com/KhronosGroup/glTF/blob/master/specification/1.0/README.md#Skins (Accessed: 01.01.2021).

[Luo 2019] Luo, G.; Deng, Z.; Jin, X.; Zhao, X.; Zeng, W.; Xie W.; Seo, H.: 3D Mesh Animation Compression based on Adaptive Spatio-temporal Segmentation, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2019.

[Lloyd 1982] Lloyd, S.: Least Squares Quantization in PCM, IEEE Transactions on information theory, Vol. 28, Nr. 2, 1982.

[Meyer 2010] Meyer, Q.; Süßmuth, J.; Sußner, G.; Stamminger, M.; Greiner, G.: On Floating-Point Normal Vectors, Computer Graphics Forum 29, 4 (2010), 1405–1409.

[Meyer 2011] Meyer, Q.; Sußner, G.; Greiner, G.; Stamminger, M.: Adaptive Level-of-Precision for GPU-Rendering, Proceedings of the Vision, Modeling, and Visualization Workshop 2011, pp. 169–176.

[NVIDIA 2021] NVIDIA Corporation: CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (Accessed: 22.01.2021).

[Purnomo 2005] Purnomo, B.; Bilodeau, J.; Cohen, J.; Kumar, S.: Hardware-compatible vertex compression using quantization and simplification, Graphics Hardware 2005, pp. 53–62.

[Sattler 2005] Sattler M.; Sarlette, R.; Klein, R.: Simple and efficient compression of animation sequences, Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2005.

[Stein 1966] Stein, P.: A Note on the Volume of a Simplex, Mathematical Association of America, 1996.

[Unity 2017] Unity Technologies: Project Quality Settings Documentation, https://docs.unity3d.com/Manual/class-QualitySettings.html#BlendWeights (Accessed: 01.01.2021).

[Volkov 2016] Volkov, V.: Understanding Latency Hiding on GPUs, University of California, Berkeley, 2016.

Ehrenwörtliche Erklärung

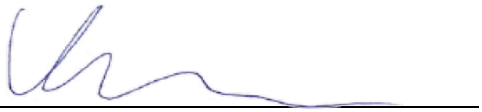
Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Titel

Komprimierung von Vertex-Blending-Attributen

selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie nicht an anderer Stelle als Prüfungsarbeit vorgelegt habe.

Coburg
Ort

27.01.2021
Datum


Unterschrift