TU WIEN Informatics

Vertex-Komprimierung mit Mesh Shadern an Modellen mit Skelettanimation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Gerald Kimmersdorfer
Matrikelnummer 01326608

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr. techn. Michael Wimmer
Mitwirkung: Projektass. Dipl.-Ing. Johannes Unterguggenberger, BSc
Projektass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Wien, 12. Oktober 2023

Gerald Kimmersdorfer	Michael Wimmer

TU WIEN Informatics

Vertex Compression with Mesh Shaders for Skinned Meshes

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Gerald Kimmersdorfer
Registration Number 01326608

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr. techn. Michael Wimmer
Assistance: Projektass. Dipl.-Ing. Johannes Unterguggenberger, BSc
Projektass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Vienna, October 12, 2023

Gerald Kimmersdorfer | Michael Wimmer

Erklärung zur Verfassung der Arbeit

Gerald Kimmersdorfer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Oktober 2023

Gerald Kimmersdorfer

Danksagung

Ich möchte meinen Betreuern Prof. Michael Wimmer, Johannes Unterguggenberger und Bernhard Kerbl meinen Dank aussprechen, dass sie mir die Möglichkeit gegeben haben, ein so interessantes Forschungsfeld für meine Bachelorarbeit zu erkunden. Besonders dankbar bin ich für ihre wertvollen Anregungen und ihre Unterstützung bei diesem besonderen Projekt. Zusätzlich möchte ich Prof. Manuela Waldner für ihre Ermutigung sowie für die Bereitstellung der dafür notwendigen Zeit danken.

Ich möchte mich auch bei der Forschungsgruppe für Computergrafik dafür bedanken, dass sie mir eine GPU zur Verfügung gestellt haben die für die Nutzung der Meshlet-Pipeline unerlässlich war.

Darüber hinaus bin ich allen Künstlern dankbar, die ihre kreativen Werke in der public domain oder unter einer Form von Attribution-Lizenz veröffentlichen. Das hat es mir ermöglicht, geeignete Modelle zu finden, um meine Implementierungen zu testen. Gleichermaßen werde ich meine Ergebnisse und meinen Code unter einer geeigneten Open-Source-Lizenz veröffentlichen, in der Hoffnung, dass sie anderen nützlich sein könnten. Insbesondere die modulare Implementierung des *Meshlet Playground* könnte für jene von Interesse sein, die neue Ansätze für die Attribut-Komprimierung oder die Meshlet-Generierung erforschen möchten.

Ein besonderer Dank gilt Christoph Peters, Bastian Kuth und Quirin Meyer für ihre hervorragende wissenschaftlichen Arbeiten in dem Bereich der Vertex-Attribut-Komprimierung, der einen Grundstein meiner Arbeit bildet. Ich möchte auch Arseny Kapoulkine meinen Dank aussprechen, der mit meshoptimizer und der Bereitstellung zahlreicher Tutorials und Ressourcen einen außergewöhnlichen Beitrag zur Welt der Computergrafik leistet.

Abschließend möchte ich meiner Familie für ihre unerschütterliche Unterstützung über all die Jahre hinweg danken, ebenso wie meinen Freunden, insbesondere Sonja Morzycki und Konstantin Kueffner, die mich in jeder Phase begleitet und beim Korrekturlesen dieser Arbeit geholfen haben.

Acknowledgements

I would like to extend my gratitude to my supervisors, Johannes Unterguggenberger and Bernhard Kerbl as well as my advisor Prof. Michael Wimmer for offering me the opportunity to explore such an interesting field of study for my Bachelor's thesis. I am particularly grateful for their valuable input and support throughout this journey. Additionally, I want to thank Prof. Manuela Waldner for her encouragement to complete the work I started and for providing me with the necessary time and resources to do so.

I would also like to thank the Research Unit of Computer Graphics for lending me the equipment, which was essential for utilizing and studying the mesh shader-based pipeline.

Moreover, I am grateful to all artists who publish their creative work in the public domain or under some form of attribution license. This enabled me to find appropriate models to test my implementations. In the same spirit of openness, I will publish my findings and code under a suitable open-source license, hoping it might benefit others. Particularly, the modular implementation of the proposed software may be of interest to those experimenting with new approaches for attribute compression or meshlet generation.

A special acknowledgment goes to Christoph Peters, Bastian Kuth, and Quirin Meyer for their exemplary work on the permutation coding algorithm, which forms the cornerstone of my research. Similarly, I extend my gratitude to Arseny Kapoulkine for his contributions to meshoptimizer and for providing numerous tutorials and resources for public use.

Lastly, I want to express my heartfelt thanks to my family for their unwavering support over the years, and to my friends, especially Sonja Morzycki and Konstantin Kueffner, who stood by me at every step and assisted in the finalization of this thesis.

Kurzfassung

Die Komprimierung von Vertex-Attributen ist eine wichtige Technik in der modernen Computergrafik, insbesondere zur Verbesserung der Leistung von Echtzeitanwendungen. In dieser Arbeit untersuchen wir aktuelle Methoden zur Komprimierung von Positionen, Normal-Vektoren, Texturkoordinaten und Blend-Attributen. Unser Hauptaugenmerk liegt auf der Komprimierung von Blend-Attributen bei Modellen mit Skelettenanimation. Wir nutzen eine neuartige Hardware-Entwicklung: die Mesh Shading Pipeline. Diese Pipeline ermöglicht es uns ein Komprimierungsschema für Blend-Attribute vorzustellen, welches eine signifikante Reduzierung des Speicherbedarfs um bis zu 92,75% im Vergleich zu bestehenden Methoden mit traditioneller Rendering-Pipeline erreicht. Darüber hinaus diskutieren und vergleichen wir verschiedene Meshlet-Erstellungsalgorithmen, Meshlet-Datenstrukturen und Meshlet-Erweiterungen innerhalb des Vulkan-Frameworks. Unsere Auswahl und die vorgeschlagenen Codecs validieren wir durch eine Reihe von Benchmarks, die sich auf Ressourcennutzung und Leistung konzentrieren.

Abstract

Vertex compression helps to enhance the performance of real-time rendering applications, making it a valuable technique in modern computer graphics. In this work, we investigate current state-of-the-art methods for the compression of positions, normals, texture coordinates and blend attributes. Our primary objective is to efficiently compress blend attributes in rigged meshes, particularly focusing on bone weights and indices. We leverage a recent hardware advancement: the mesh shading pipeline. This pipeline enables us to propose a novel compression scheme for blend attributes, which achieves a significant reduction in memory usage of up to 92.75% compared to existing state-of-the-art methods using a traditional rendering pipeline. Additionally, we briefly discuss and compare different meshlet building algorithms, meshlet buffer structures, and meshlet extensions within the Vulkan framework. Finally, the proposed codecs are validated through a series of benchmarks focused on resource utilization and performance.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Mesh Shading 1
1.2 Linear Blend Skinning (LBS) 3
1.3 Research Objective 4

2 Methodology **5**
2.1 Vertex Attribute Compression 5
2.2 Meshlet Difference Encoding 15

3 Implementation **19**
3.1 Mesh Shading Pipeline 20
3.2 Meshlet Building 21
3.3 Compression Codecs and Buffer Layouts 24

4 Evaluation **29**
4.1 Environment and Datasets 29
4.2 Memory Consumption 31
4.3 Performance 33

5 Conclusion **35**
5.1 Future Work 35

6 Appendix **37**

List of Figures **39**

List of Tables **41**

List of Algorithms **43**

CHAPTER 1

Introduction

The geometric detail in rendering applications is continuously increasing, with models in photorealistic renderings now containing billions of triangles. This trend is pushing VRAM throughput to its limits, as the required bandwidth must also increase to efficiently transfer model data from the disk or within the GPU memory. Consequently, geometry compression techniques have become critical to meeting these demands [MLDH15].

In this thesis, we present methods to address this issue by compressing vertex attributes in skinned meshes. The primary contribution of this work is a novel compression scheme for blend attributes when using a mesh-shader based pipeline. We introduce a difference encoding technique that utilizes the local similarities of bone weights and bone indices within local patches of the model. This approach achieves significant memory savings while maintaining competitive rendering performance.

This chapter serves to briefly introduce the key concepts of this thesis and outlines the general research objective for this work.

1.1 Mesh Shading

Mesh shading represents a significant evolution in the field of 3D graphics, offering a stark contrast to the traditional *vertex shading pipeline* (Figure 1.1). Over the years the traditional pipeline got extended by multiple specialized stages like the tessellation and the geometry stage leaving the whole pipeline in a rather complex state. The Task/Mesh pipeline intends to offer a more streamlined approach, resulting in greater flexibility while still utilizing hardware-accelerated functionality, like rasterization.

In traditional vertex shading, each vertex is processed individually. In contrast, a Mesh shading pipeline can process a group of vertices simultaneously within the *Mesh Shader*. These clusters of vertices are known as *meshlets* (see Figure 1.2).

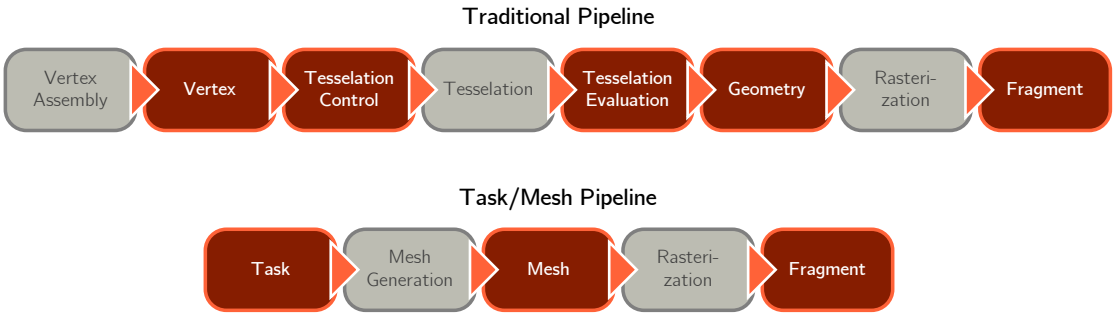


Figure 1.1: This figure illustrates the distinctions between the two geometry pipelines available in modern GPUs. Grey boxes represent fixed-function stages, while the red stages are fully customisable through the definition of appropriate shader programs [Kub18].

An optional *task shader* can control the invocations of the Mesh Shader, potentially calling for the additional rendering of meshlets. This functionality makes it particularly suitable for the premature culling of geometry [UKPW21] and Level of Detail (LOD) selection [Eng20]. Particularly for culling purposes, properly creating meshlets from existing geometry is a crucial step to ensure peak performance when rendering the data [JFB23].

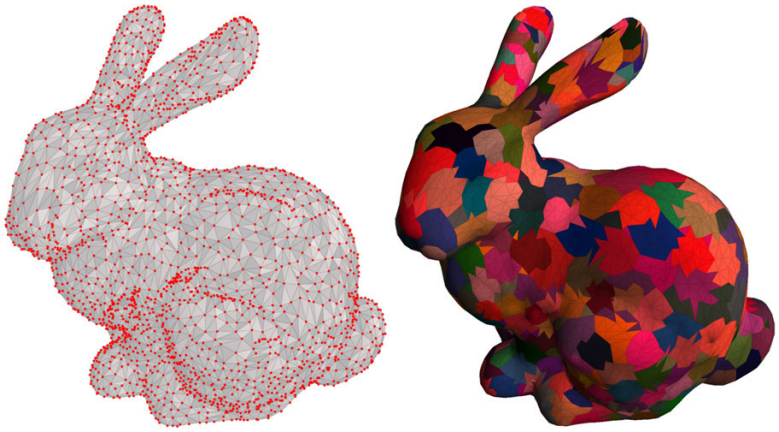


Figure 1.2: In the traditional rendering pipeline, each vertex (red dots in left image) of the model is individually processed by the vertex shader. In a mesh shader-based pipeline, groups of vertices (right image) share a common storage structure and allow for efficient neighborhood access, which facilitates the use of more advanced compression algorithms.

Additionally, the shared memory buffer for a group of vertices makes this rendering technique well-suited for advanced compression techniques. Recent research shows that by proper indexing of the triangle mesh data, compression rates of up to 16:1 for index buffer sizes can be achieved [MSS24, KOK+23]. Compressing vertex data by utilizing

local clusters is also fairly old [LCL10], but is and will be further revitalized in real-time rendering software having the proper hardware support [BBM24].

A comparable approach, serving as a best-practice example of efficient cluster compression, is Epic's Nanite system, as described by Karis et al. [KMWM21]. Their system employs custom compute shaders to perform efficient culling on clusters containing approximately 128 triangles. Compressed clusters are streamed from the disk and decoded on the GPU with an adjustable bit rate.

1.2 Linear Blend Skinning (LBS)

Over the years, LBS has been known by many names, including skeleton subspace deformation, enveloping, vertex blending, smooth skinning, bones skinning or linear blend skinning [LD12]. It is a widely used technique for animating and rendering organic entities, such as humans and animals. It is prevalent in both the gaming and film industries and is often coupled with motion capture technologies. The technique excels in animations that adhere to a hierarchical structure, making it suitable for animating rigid bodies such as cars or robots.

Mesh skinning generally enables the deformation of a mesh by applying a series of transformation matrices to an underlying structure. Typically, those transformation matrices are derived from a set of hierarchically aligned interconnected parts. Those parts are commonly known as *bones*. The set of those bones form the skeleton of the mesh, as visible in Figure 1.3.

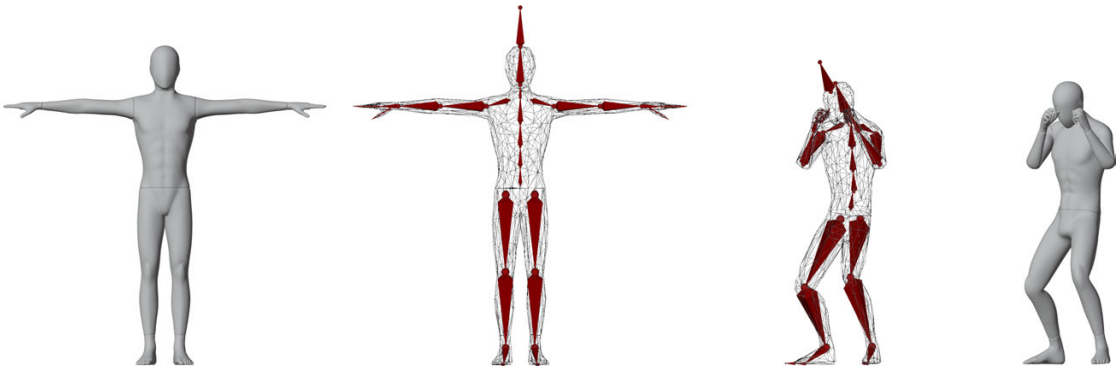


Figure 1.3: The Mannequin character from the Mixamo Collection consists of 65 bones. Every vertex of the mesh is influenced by a subset of those bones with different magnitudes (bone weights).

Each vertex within a mesh is algorithmically or manually associated with a subset of the skeleton's bones. The IDs of those bones are referred to as the *bone indices*. Additionally, a factor is associated with every vertex-to-bone connection, representing the degree to which the bone influences the vertex's position. These factors are known as *bone weights*.

To calculate the final position of each vertex, we apply the transformations of each bone influencing the vertex, weighted by their respective influence on the vertex. This process sums up the effects of all relevant bones on the vertex's initial position to determine its final position. Mathematically, this can be described as:

$$v_i^t = \sum_{j \in B_i} w_{ij} (T_j^t p_i)$$

where v_i^t is the deformed position of the i-th vertex at time t, w_{ij} is the influence of the j-th bone on the i-th vertex, p_i is the position of the i-th vertex in the rest pose, and B_i is the set of bones influencing the i-th vertex (bone indices). Here, T_j^t represents the general transformation matrix of the j-th bone at time t.

In the context of real-time rendering, the number of bones that can influence a single vertex is often capped at four, known as the *sparseness constraint*. This requirement is imposed for performance reasons, as the calculation of the transformation matrix for each vertex is linearly dependent on the number of influencing bones. Additional bones necessitate more data transfer to the GPU and potentially more expensive matrix multiplications.

1.3 Research Objective

In real-world scenarios, vertex attributes for Linear Blend Skinning (LBS) models exhibit significant similarity to vertices in close proximity. This phenomenon is illustrated in Figure 1.4, where bone weights and unique bone indices are visualized for two different models. As meshlets are generally generated as small local patches, the variation in bone weights and bone indices for vertices within a meshlet is often minimal or even non-existent. This observation motivates the development of a compression scheme that leverages this geometric information.

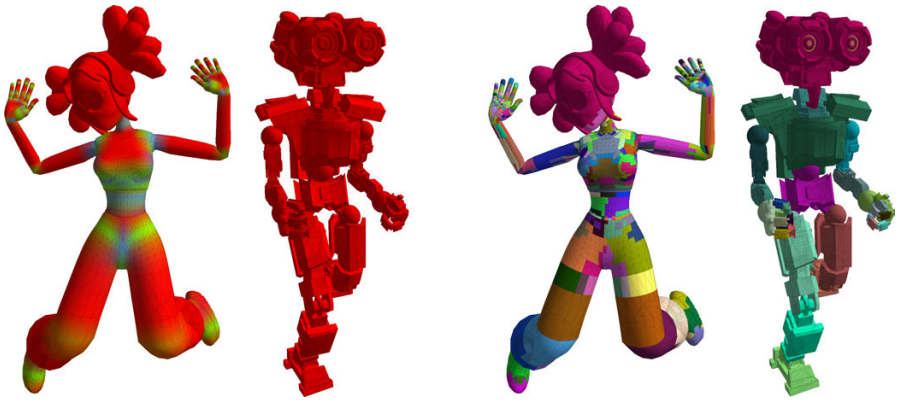


Figure 1.4: The left images depict the scaled values of the first three bone weights (by equation 2.19) encoded as RGB colors. The right images show all unique bone index vectors in the `Michelle` and `Robot` model.

CHAPTER 2

Methodology

In section 2.1 the compression methods used for the various vertex attributes are discussed. The methods described in this section can be used in either a traditional or mesh shader-based pipeline, as they completely operate locally on the individual data per vertex. Those compression techniques act as the baseline for our further investigation. Section 2.2 summarizes how meshlets can be used to further reduce the necessary storage for our bone weights and bone indices.

2.1 Vertex Attribute Compression

Vertex attributes are properties associated with the vertices of a model. Prominent examples include position and normal vectors. When dealing with rigged meshes, these attributes need to be extended by bone indices and bone weights, representing which bones affect the position and normal of each vertex. Since vertex data constitutes a significant portion of the total model data, it is crucial to find compression methods that reduce their memory footprint. These methods are typically categorized under the term *vertex compression*. Although various methods exist for compressing vertex attributes by reducing vertices, bones, or recalculating bone weights during pre-processing steps, [PBCK05] this paper focuses on ways to compress the data without mutating the vertex list and without applying significant changes to their values, aside from the effects of the lossy nature of the investigated and proposed algorithms.

2.1.1 Positions

The position of a vertex is usually given in local coordinates described by three floating point numbers. With a usual bit-depth of 32-bit per float, we end up with 12 bytes per vertex to store the vertex position. A common way to compress these is by using Quantisation.

The first step is to normalize the position values. To this end, the maximum ($p_{\max}$) and minimum ($p_{\min}$) bounds of all the vertex positions are evaluated for a given model. Knowing those values, a normalized position with all component values in between $[0, 1]$ can be calculated:

$$\Delta p = p_{\max} - p_{\min} \qquad p_{\text{norm}} = \frac{p - p_{\min}}{\Delta p} \tag{2.1}$$

We calculate an unsigned int vector $\tilde{p}$ with a bit-depth of b using the following formula:

$$\tilde{p} = \left\lfloor p_{\text{norm}} \cdot (2^b - 1) \right\rfloor \tag{2.2}$$

To decode the original position we are required to store $p_{\min}$ and Δp as a property of the model. Knowing those values we can decode $\tilde{p}$ using the inverse operation:

$$p'_{\text{norm}} = \frac{\tilde{p}}{2^b - 1} \qquad p' = p'_{\text{norm}} \cdot (\Delta p) + p_{\min} \tag{2.3}$$

This method works best for models where the vertices are uniformly distributed. It is a lossy compression mostly depending on the bit-depth b.

To calculate a *scale-invariant* error we use the normalized positions to end up with the following formula for the mean $\mathcal{L}^2$-distance error over all vertices N:

$$\text{Mean } \mathcal{L}^2\text{-d error} = \frac{1}{N} \sum_{i=1}^{N} \|p'_{\text{norm}} - p_{\text{norm}}\| \tag{2.4}$$

Table 2.1: Scale-invariant quantization error for different bit depths

Model	#Vertices	Mean $\mathcal{L}^2$-d error			
		$b = 8$	$b = 10$	$b = 16$	$b = 21$
Stanford Bunny	208,353	3.78×10^{-3}	9.39×10^{-4}	1.46×10^{-5}	4.32×10^{-7}
Michelle	56,542	3.74×10^{-3}	9.31×10^{-4}	1.46×10^{-5}	4.24×10^{-7}
Mixamo Group	624,127	3.76×10^{-3}	9.38×10^{-4}	1.46×10^{-5}	4.22×10^{-7}
Porsche	597,003	3.76×10^{-3}	9.36×10^{-4}	1.46×10^{-5}	4.28×10^{-7}

More elaborate solutions to minimize the error and the required bit-depth can be determined with a more complex prediction scheme [CM02] or by splitting the geometry into locally closer patches [LCL10].

2.1.2 Texture Coordinates

Texture coordinates, commonly known as UV coordinates, are often compressed using quantization, similar to positional data. These coordinates are typically normalized,

meaning they fall within the range $[0, 1]$ for both the U and V axes. In systems that strictly enforce this constraint, quantization is straightforward, involving a simple multiplication by 2^b, where b is the number of bits used for quantization. In such systems, b can be chosen such that 2^b equals the maximum allowed texture size without losing significant details or creating visual artifacts.

In the more general case, where this constraint is not enforced, it is necessary to scale the u and v values similarly to how positions are handled, ensuring that values greater than 1 or less than 0 are properly decoded. In these cases, b must be selected larger than the maximum allowed texture size to accommodate the extended range of values. This ensures that the quantization process maintains the necessary precision for UV coordinates that fall outside the standard $[0, 1]$ range, thereby avoiding significant loss of detail or visual artifacts when the texture is applied.

2.1.3 Normals

Like positions, normals are naively stored as 3-component float vectors. Discretizing this value leads to many possible bit patterns being wasted [CDE+14]. This can be seen in Figure 2.2 where this approach (referred to as the Euclidean Decode Method) yields far fewer possible normal encodings with similar memory consumption compared to other methods.

Given the constraint on normalized vectors:

$$x^2 + y^2 + z^2 = 1 \tag{2.5}$$

it becomes evident that all possible normal vectors are positioned on the unit sphere with radius $r = 1$. Therefore a common approach is to store the normals as *spherical coordinates*. Without the need to save r explicitly, we only need a 2-component float vector per normal saving 33% of the necessary storage. The conversion from Cartesian coordinates (x, y, z) to spherical coordinates (θ, ϕ), given the fact that $r = 1$ is done using the following formulas:

$$\theta = \arctan2(y, x) \qquad \phi = \arccos(z) \tag{2.6}$$

with $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. To decode back to Cartesian coordinates, we use:

$$x = \sin(\phi)\cos(\theta) \qquad y = \sin(\phi)\sin(\theta) \qquad z = \cos(\phi) \tag{2.7}$$

This approach has two downsides:

1. Uniformly spaced spherical coordinates are badly distributed over the unit sphere and clump together at the poles (see Figure 2.2).
2. Trigonometric functions are expensive on some hardware, but required for Decoding.

Better results are promised by the use of *octahedral encoded unit vectors* [MSS+10]. The idea is to project the sphere onto an octahedron and unfold it onto a unit square. This is done by reflecting the faces on the negative z-axis over the appropriate diagonals (see Figure 2.1).

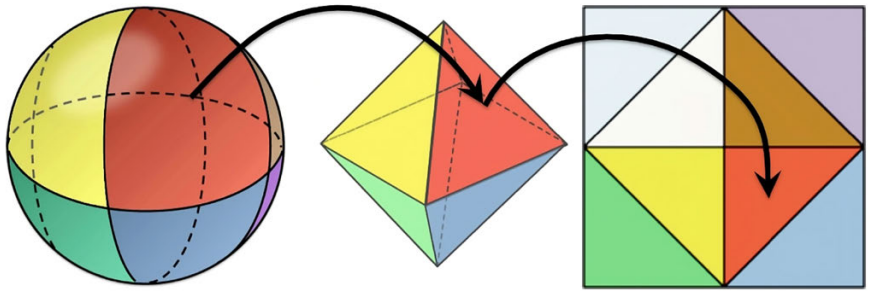


Figure 2.1: The octahedron encoding is defined by a mapping of the octants of a unit sphere onto the faces of an octahedron. The faces are mapped onto a unit square which can be stored in two components [CDE+14].

This algorithm has an almost perfect distribution of values over the unit sphere, as visible in Figure 2.2. Furthermore, it is computationally very efficient to solve. To encode a normal n we have to:

Normalize the vector using the Manhattan norm:

$$n' = \frac{n}{|n_x| + |n_y| + |n_z|} \tag{2.8}$$

The values of n' for $n_z >= 0$ directly correspond to the mapped octahedron coordinates. For the case $n_z < 0$ we need to flip the faces as depicted in Figure 2.1 resulting in:

$$f = \begin{cases} (n'_x, n'_y) & \text{if } n'_z \geq 0 \\ (1 - |(n'_y, n'_x)|) \cdot \text{sign}(n'_y, n'_x) & \text{if } n'_z < 0 \end{cases} \tag{2.9}$$

where the sign, $||$ and $\cdot$ operate on the individual components of the 2-d vectors and f represents the 2-dimensional octahedron encoded normal vector, with $f_x, f_y \in [-1, 1]$.

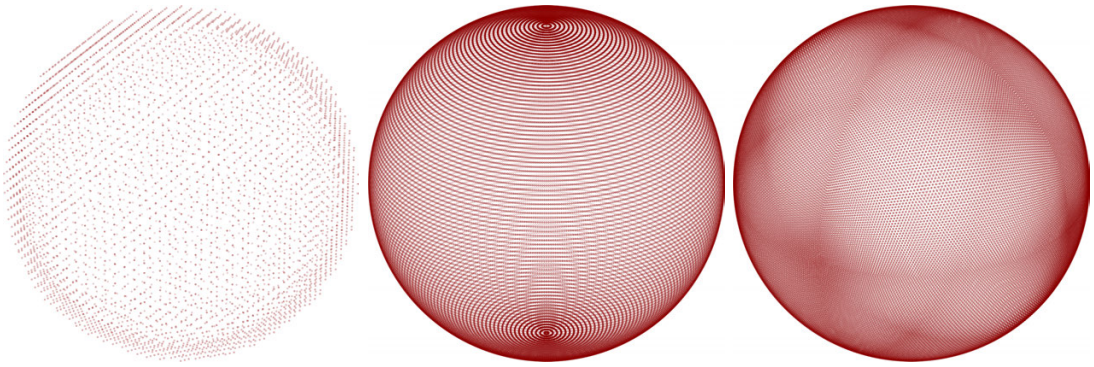
In order to decode f, the formulas for the individual components of $n' = (n'_x, n'_y, n'_z)$ can be derived as:

$$t = \max(-1 + |f_x| + |f_y|, 0) \tag{2.10}$$
$$n'_x = n_x + \text{sign}(n_x) \cdot t \tag{2.11}$$
$$n'_y = n_y + \text{sign}(n_y) \cdot t \tag{2.12}$$
$$n'_z = 1 - |f_x| - |f_y| \tag{2.13}$$

After normalization:

$$n' = \frac{n'}{||n'||} \tag{2.14}$$

n' now holds the decoded components of f, with $n' \approx n$



(a) Euclidean (b) Spherical (c) Octahedron Encoding

Figure 2.2: Comparison of the distribution of unit vectors using different encoding methods. The naive quantized Euclidean method depicted here uses 5 bits per channel, resulting in a total of 15 bits. In contrast, methods (b) and (c) use 8 bits per channel resulting in a total of 16 bits.

The appropriate quantization of f is done in the same fashion as for the positions. In terms of bit depth, we choose $b = 16$ so that we can pack the normals into a single 32-bit unsigned integer. A comparison of the errors for the different tested methods can be seen in Table 2.2, highlighting the octahedron method as the best fit for this purpose.

Table 2.2: Errors for different normal encoding schemes on various datasets. The encoded data fits into 32-bit leaving 10-bit for the components of the Euclidean method and 16-bit for the components of the spherical and octahedral methods.

Model	#Vertices	Mean $\mathcal{L}^2$-d error		
		Euclidean	Spherical	Octahedron
Stanford Bunny	208,353	9.40×10^{-4}	2.48×10^{-5}	2.16×10^{-5}
Michelle	56,542	9.54×10^{-4}	2.38×10^{-5}	2.09×10^{-5}
Mixamo Group	624,127	9.47×10^{-4}	2.90×10^{-5}	2.62×10^{-5}
Porsche	597,003	9.50×10^{-4}	2.47×10^{-5}	2.05×10^{-5}

2.1.4 Bone-Indices

Given the common sparseness constraint with rigged meshes, every vertex is influenced by at most four bones. This necessitates storing four unsigned integers per vertex, which act as lookup indices inside the array of bone matrices. The required bit-depth of these integers significantly depends on implementation decisions. In many cases, 8-bit per

model is sufficient, but generally, 16-bit is a more robust choice. This choice ensures that the system can handle larger models and avoid potential overflow issues.

Kuth et al. [KM21] enhanced compression by noting that only a limited number of bone combinations are utilized, due to the tendency of neighboring vertices to share the same or similar bones of influence (see Figure 2.3). These distinct combinations, referred to as *unique tuples*, are stored in a separate table, with only a lookup ID (referred to as *tuple index*) remaining as a vertex attribute.

[a] No Compression *(406 unique tuples)* [b] With tuple reuse *(263 unique tuples)* [c] Our solution *(94 unique tuples)* [d] Meshlets *(886 Meshlets)*

Figure 2.3: Comparison of the amount and distribution of distinct tuple indices for different methods applied to the Mixamo `Michelle` character. We propose a sorting and merging step to reduce the number of unique tuples in the lookup table. The meshlets shown in [d] illustrate that most meshlets share the same tuple index.

Peters et al. [PKM22] introduced two novel optimizations to this approach:

1. If only a single bone affects the vertex, it is directly encoded as the tuple index.
2. Tuple Reuse: For vertices influenced by fewer than four bones, their tuple may be replaced by one containing the same bones in the same order. As the weight for any additional bones is zero, they do not affect the calculation of the final vertex position.

Rule 1 was not implemented in our work due to the following considerations:

- No extra handling of vertices at decompression time is necessary.
- The difference-based encoding algorithm works best if tuple indices vary as little as possible. Therefore it's generally better if the singular tuples have a shared tuple index as opposed to an arbitrary bone index.

- The bit-depth for the tuple index must be at least the same size as for the bone indices. If there are no unused bones in the model this will typically be the case, but scenarios can be constructed where this constraint could pose a problem, such as having a shared bone buffer in the scene but individual lookup tables.
- The size of the lookup table is only minimally affected by this rule, as the corresponding tuple—containing only one bone—will usually be replaced by a different tuple when applying Rule 2.

To implement Rule 2, we first set all irrelevant bone indices to the highest possible number (denoted as ∞ in Figure 2.4). Subsequently, the list of tuples is organised in descending order, ensuring that tuples that can be replaced by others appear before them. The last value of these emerging groups is then saved as an entry in the final Lookup-Table.

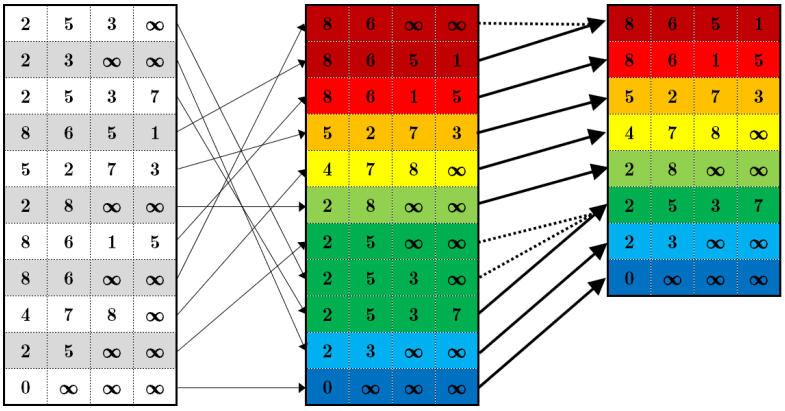


Figure 2.4: Tuple-Reuse: Irrelevant indices are symbolized as ∞. The lookup table is then sorted, and only the last tuple of each corresponding group is saved [PKM22].

Since the tuple indices of the vertices have now changed, it is necessary to update the attributes accordingly. It is advisable to log all changes made to the lookup table in a hash map and apply them all at once.

Index-Sorting and Merging

To further reduce the size of the lookup table and consequently the number of unique tuple indices, we propose an additional pre-sorting of the tuples. The applied permutation can be stored either within the bone weights or, alternatively, as a separate permutation index. Given our constraint of 4 bones, there are $4! = 24$ possible permutations, which requires 5 bits per vertex to store the permutation.

Subsequently, we apply the same tuple-reuse technique as described earlier. By storing the permutation for each vertex individually, we can further merge tuples that contain irrelevant indices. Our greedy algorithm proceeds with the following steps:

1. Select the entry with the highest number of irrelevant indices.

2. Search for the optimal merge candidate based on two criteria:

   a) Minimize the number of irrelevant indices.

   b) If criterion 2a applies to multiple candidates, we choose the one that maximizes the number of shared bone indices. This heuristic may result in similar tuple indices for vertices located near each other.

3. Merge the tuples by appending the relevant bone indices of the selected merge candidate to the end of the entry selected in step 1. Adjust the permutation and tuple indices for the vertices of the merge candidate accordingly.

4. Repeat from step 1 as long as possible combinations remain.

An example of the sort and merge process is illustrated in Figure 2.5. In the initial step, we sort the bone indices within their respective tuples. Next, we apply the tuple reuse technique as proposed by Peters et al. [PKM22] on this modified table. In the final step, we merge tuples that contain a sufficient number of irrelevant indices. In this example, we first select the tuple $0 \mid \infty \mid \infty \mid \infty$, as it contains the highest number of irrelevant values. The only merge candidate satisfying the primary criterion is $4 \mid 7 \mid 8 \mid \infty$. Consequently, we merge these entries by appending the candidate to the end of the selected tuple. Further combinations are still possible, so we choose $6 \mid 8 \mid \infty \mid \infty$ as the next tuple to merge, given that it has the highest number of irrelevant indices among the remaining tuples. According to the second criterion, we merge it with $2 \mid 8 \mid \infty \mid \infty$, as both share the bone with index 8. With only $2 \mid 5 \mid \infty \mid \infty$ left, there is no additional merge candidate and the algorithm terminates.

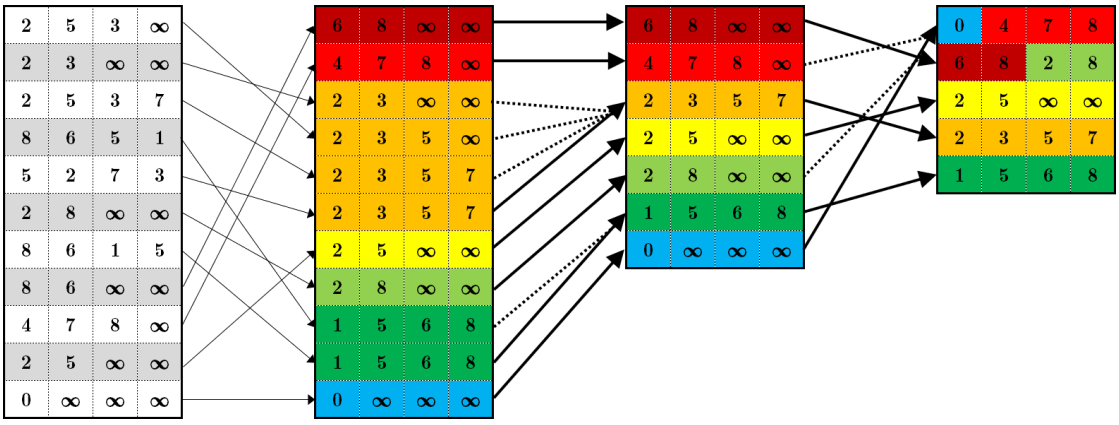


Figure 2.5: Sorting and merging the bone indices allows for a better compression rate of the lookup table.

2.1.5 Bone-Weights

Bone weights are stored as vectors of floating-point numbers. These vectors adhere to the following constraints:

- *Sparseness*: Limits the dimensionality. We consider bone weights only up to a dimensionality of 4.
- *Non-negativity*: All weights are positive.

$$w_j \geq 0 \quad j \in [1, N] \tag{2.15}$$

  where w_j represents the weight of the j-th bone for a single vertex and N the dimension of the bone weight vector.

- *Affinity*: The sum of the weights for all bones associated with a single vertex must be equal to 1:

$$\sum_{j=1}^{N} w_j = 1 \tag{2.16}$$

These constraints offer significant possibilities for compression. Equation 2.15, paired with Equation 2.16, ensures that all bone weights are between zero and one. This allows for easy quantification of these attributes. Additionally, Equation 2.16 enables the elimination of one bone weight, as it can be reconstructed from the others using:

$$w_j = 1 - \sum_{\substack{k=1 \\ k \neq j}}^{N} w_k \tag{2.17}$$

The order in which bone weights are stored is arbitrary as long as they correspond with the appropriate bone index. However, it is generally advised to sort them to allow for premature termination of the final vertex calculation. Additionally, the resulting constraint provides further compression possibilities:

$$w_1 \geq w_2 \geq w_3 \geq w_4 \tag{2.18}$$

Knowing that we have a sorted list of weights, we choose not to explicitly store the largest one w_1. The remaining inequalities $w_2 \geq w_3 \geq w_4$ form a 3-dimensional simplex (or tetrahedron) in weight space. Naively quantizing those weights between $[0, 1]$ would yield many invalid tuples, as can be seen in Figure 2.6 (a).

To cover more space inside the unit cube, the simplex can be stretched with a simple consideration: The weight at position j is always smaller than $\frac{1}{j}$. This is a direct result of the constraint applied by sorting the elements (Equation 2.18) and by the affinity

constraint (Equation 2.16). A proof by contradiction is trivial and provided in Section 6.1. Hence, we can stretch the weights by:

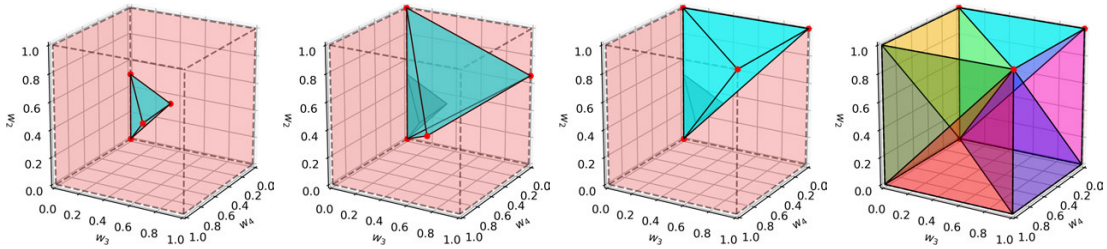
$$w'_2 = 2w_2 \quad w'_3 = 3w_3 \quad w'_4 = 4w_4 \tag{2.19}$$

which is visualised in Figure 2.6 (b). This tetrahedron now covers $\frac{1}{6}$ of the whole weight space but loses the original order, such that Equation 2.18 is not applicable for w' anymore.

To that end, Peters et al. [PKM22] proposed a different approach to scale the weights:

$$w'_2 = 2w_2 \quad w'_3 = 3w_3 + w_2 \quad w'_4 = 4w_4 + w_2 + w_3 \tag{2.20}$$

which is visualised in Figure 2.6 (c). This approach ensures that the weights w' are still ordered, meaning $w'_2 \geq w'_3 \geq w'_4$.



(a) Base (b) Eq. 2.19 applied (c) Eq. 2.20 applied (d) Permutations

Figure 2.6: All possible combinations of bone weights lie inside a 3-dimensional simplex (a). A naive linear transformation can be applied to cover $\frac{1}{6}$ of all possible encodings (b). Peters et al. [PKM22] proposed a different linear transformation which is order-preserving (c). By shuffling the weights after applying Eq. 2.20 we can reach all possible values inside the unit cube. All possible permutations are shown in different colors (d).

The original weights can be reconstructed using:

$$w_2 = \frac{w'_2}{2} \quad w_3 = \frac{2w'_3 - w'_2}{6} \quad w_4 = \frac{3w'_4 - w'_2 - w'_3}{12} \tag{2.21}$$

Another useful property of w' is that by shuffling its components it is possible to attain any other point inside the unit cube. The authors proposed to select a permutation based on some bits of the tuple index, effectively saving another $\log_2((N-1)!) = \log_2(3!) = 2.6$ bits. (Considerably more for $N > 4$)

One edge case still has to be considered: Components of w' can be equal in which case multiple permutations could be valid for the resulting weight vector and the reconstruction of the tuple index would be impossible. To eliminate this problem the authors propose to push the quantized values (a_i) apart to force a strict ordering, meaning $a_2 > a_3 > a_4$.

They furthermore introduced precision factors B_i to allow for extra bits to be used on a single weights basis. This allows for fine-tuning the compression quality. Finally, they

pack the quantized values together with the rest of the payload into one unsigned integer. The other part of the payload is saved inside the permutation.

Additional information are given by Peters et al. [PKM22]. They document implementation details and pitfalls, while also providing ready-to-use code modules[1] for GLSL and C-language.

A different approach is proposed by Kuth et al. [KM21], who suggests assigning index numbers to all possible and valid points inside the tetrahedron. Instead of storing the weight vector, only this index is stored. This scheme is named *Optimal Simplex Sampling*. However, due to the expensive decoding algorithm, which requires solving polynomial equations, Permutation Coding yields better results in our tests. Consequently, we chose this compression scheme to pair with our Difference Encoding algorithm.

2.2 Meshlet Difference Encoding

Given the strong local similarity for bone attributes, as presented in Section 1.3, we propose storing only the differences from a reference value within the vertex attributes. This reference value is stored in the meshlet data along with a resolution value. The resolution value defines how many bytes are necessary to properly store the individual difference values.

To calculate the reference value b_m for a given meshlet m, we take the minimum of all bone attributes b_{im} for that meshlet:

$$b_m = \min(b_{im})$$

The difference Δb_{im} for each vertex i is then calculated as:

$$\Delta b_{im} = b_{im} - b_m$$

The bone attribute b_{im} for a given vertex i from a meshlet m can then be evaluated on the GPU with a simple addition.

To limit the number of cases we must handle, we define r_m, the bit resolution for meshlet m, in such a way that varying the padding parameter p results in different paddings.

$$r_m = \left\lceil \frac{\log_2(\max(\Delta b_{im}) + 1)}{p} \right\rceil p \tag{2.22}$$

where $\max(\Delta b_{im})$ is the maximum difference for all vertices in meshlet m.

We conducted experiments with $p \in \{8, 16, 32\}$:

- $p = 8$ means that the difference is stored in 8-bit aligned data.

[1] `https://momentsingraphics.de/I3D2022.html`

- $p = 16$ means that the difference is stored in 16-bit aligned data.
- $p = 32$ is the equivalent of only having special treatment for vertices that share the same bone attribute, as those are 32 bits long.

This approach has one crucial downside: vertices can only be referenced by different meshlets if r_m and b_m are the same. To address this issue, we introduced a preprocessing step in which we create copies of the vertices affected and adapt the references accordingly.

The entire scheme is summarised in Algorithm 2.1, and a visualization of how the different bit resolutions are present in some of our models is shown in Figure 2.7.

Algorithm 2.1: Vertex Data Preparation for Meshlet Difference Encoding

```
Compress bone weights and indices with permutation codec.
foreach meshlet m do
    Calculate b_m, r_m
end
foreach vertex i referenced by multiple meshlets do
    foreach meshlet tuple m_1, m_2 referencing i do
        if b_m1 ≠ b_m2 or r_m1 ≠ r_m2 then
            Split the vertex i Update reference in meshlet m_2
        end
    end
end
foreach vertex i do
    Calculate Δb_im
end
Upload 8-bit vertex buffer with respect to r_m for all meshlets m.
foreach meshlet m do
    Change the vertex_id to the appropriate offset inside the 8-bit vertex buffer.
end
Upload additional buffer containing b_m and r_m for each meshlet m.
```

In general, this encoding scheme works with any form of vertex attribute data. It is neither bound to Permutation Coding [PKM22] nor specific to bone attributes. We also tested the potential compression possibilities when applied to position, texture coordinates, or normals. However, those results have not been promising as these attributes exhibit too great variances within the respective meshlets.

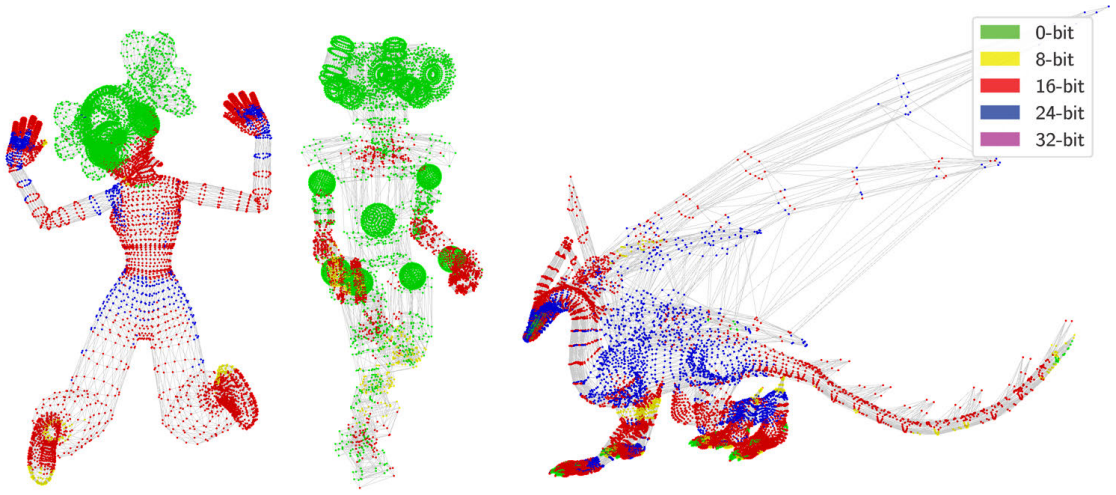


Figure 2.7: Our Meshlet Difference Encoding algorithm stores bone attributes with varying bit resolutions inside the vertex buffer. The bit resolutions are visualized with different colors on the `Michelle`, `Robot` and `Dragon` models. The bone attributes are encoded with the Permutation Coding Algorithm [PKM22].

CHAPTER 3

Implementation

To test the proposed method, a Vulkan application (see Figure 3.1) was implemented. The project is publicly available on GitHub[1] and uses the Auto-Vk-Toolkit Framework [Com24]. The software allows us to compare the efficiency of the implemented algorithms with timing and buffer sizes presented on the screen.

The processing pipeline for the model data is generally structured into four different steps:

1. *File Loading:* All meshes inside the scene are loaded into one vertex and one index array, which can optionally be optimized using the meshoptimizer library [Kap17] library. Bone weights are cleaned up, and positions and texture coordinates are rescaled to fit within the range of $[0, 1]$ (as described in Section 2.1.1).

2. *Meshlet Building:* The geometry is split into meshlets and appropriate buffers are generated. If not specified, we use the meshlet creation algorithm provided by the meshoptimizer library [Kap17]. Additional information about this step is provided in Section 3.2, where we compare two different methods for flattening and accessing the data in a GPU-friendly manner.

3. *Vertex-Compression:* A compressed vertex buffer, with optional additional meshlet buffers, is created. The compression is performed on the CPU side. We evaluate seven methods, outlining their differences in Section 3.3.

4. *Rendering:* All the required buffers are uploaded to the GPU, and the necessary shaders are compiled. We implemented a custom *Shader Meta Compiler*, which is a text-based preprocessing step for our shader files. It allows us to introduce dynamic preprocessor constants. The SPIR-V [Khr24] compiler is then executed

[1] `https://github.com/GeraldKimmersdorfer/compressed_meshlet_skinning`

Figure 3.1: The Skinned Meshlet Playground is a Vulkan application implemented specifically for this thesis. It allows us to compare our proposed algorithm to other state-of-the-art methods.

on these altered files, eliminating all unnecessary code for the selected pipeline layout. This implementation detail enables very compact shader files with less redundancy in the code. We investigated the differences between two possible Vulkan Meshlet-Extensions in Section 3.1.

3.1 Mesh Shading Pipeline

Support for the Mesh Shading pipeline was initially introduced with the NVIDIA Turing Architecture in 2018 [Kub18]. Shortly afterward, they offered a device extension named `VK_NV_mesh_shader` to provide the necessary software support for the new hardware architecture. In September 2022, Vulkan released a new vendor-independent solution for Mesh Shading called `VK_EXT_mesh_shader`[2].

[2] `https://github.com/KhronosGroup/Vulkan-Docs/blob/main/ChangeLog.adoc`

While there are some differences in the syntax[3], the general structure of the extensions is fairly similar, particularly for the functionality required for this work. The most notable difference for this project's scenario is that the NVIDIA extension supports an intrinsic function, `writePackedPrimitiveIndices4x8NV`, which allows for four 8-bit wide indices packed in a single 32-bit integer to be sent at once.

Given the layout of our redirected mesh buffer detailed in Chapter 3.2, this saves some clock cycles on the GPU, as we can immediately send the packed indices to the rasterizer. For the Vulkan extension, we implemented a workaround, which has to unpack the data first.

In the provided software we implemented support for both extensions and ran benchmarks with our test scenes to evaluate differences in frame time (see Table 3.1). In all cases, peak performance was achieved by using the NVIDIA extension. Given these results, we used the NVIDIA extension for all other benchmarks unless explicitly specified.

	NV Extension		**VK Extension**	
Meshlet-Type:	**Native**	**Redirect**	**Native**	**Redirect**
Playground	130.94 ms	**127.46 ms**	140.38 ms	129.67 ms
Mixamo G.	92.39 ms	**91.93 ms**	99.77 ms	93.24 ms
Monsters	209.10 ms	**207.24 ms**	218.96 ms	209.89 ms

Table 3.1: Performance comparison between NVIDIA and Vulkan extensions for Mesh shading by monitoring *GPU frame timings*. NVIDIAs extension reaches peak performance on our hardware. Timings are taken with a copy count of 500 instances per model and no vertex compression.

3.2 Meshlet Building

Constructing meshlets for a model often depends on specific requirements. For compact storage, different patches might be more advantageous compared to those used for rendering a mesh. We deem an approach that minimizes cache misses and maximizes vertex reuse, while selecting faces with similar normals, as optimal for most 3D models and rendering scenarios. In such cases, premature culling of the entire meshlet is more efficient. For our specific purposes, meshlets that reduce the variance of bone attributes are ideal. This implies that all vertices of a meshlet should possess similar bone weights and bone indices to achieve the best possible compression rate.

Since bone attributes constitute only a relatively small fraction of the overall model data size, a meshlet creation algorithm optimized solely for compressing these attributes will result in suboptimal selections for culling or rendering. Taking this into account, we propose our compression scheme to be generally applicable, independent of any specific meshlet-building strategy.

[3] `https://www.khronos.org/blog/mesh-shading-for-vulkan`

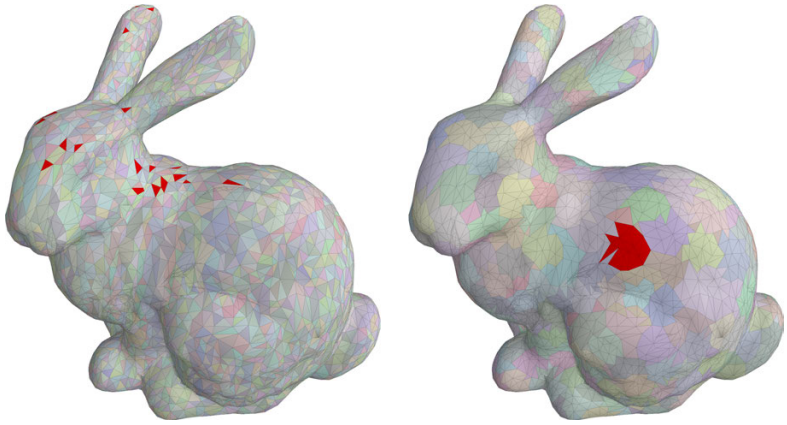


Figure 3.2: Comparison of a simple greedy meshlet generating approach (AVK-Default, left side) and an approach which takes vertex reuse, proximity and normal variance into account (Meshoptimizer, right). One random meshlet is highlighted in both images.

Partitioning a model into meshlets is an ongoing area of research [JFB23, KOK+23, MSS24] and is not the primary focus of this work. However, we briefly evaluated two different meshlet construction algorithms:

1. *AVK-Default:* This algorithm is integrated into the Auto-VK-Toolkit-Framework [Com24]. It follows a simple greedy strategy, constructing meshlets by sequentially adding triangles as they appear in the index buffer. Upon reaching the maximum allowed vertex or index count, the algorithm proceeds to the next meshlet. Consequently, the quality of the meshlets is dependent on the vertex order within the index buffer.

2. *Meshoptimizer:* The meshoptimizer framework [Kap17] offers a more sophisticated meshlet-building algorithm that maximizes vertex reuse inside meshlets while prioritizing vertex proximity and the minimization of the variance inside the vertex normals.

A visual comparison of both implementations can be seen in Figure 3.2. Unless explicitly stated otherwise, all subsequent and preceding tests, as well as benchmarks, were conducted using meshlets created by the meshoptimizer framework. In all scenarios, it generated fewer meshlets with their respective vertices in closer proximity.

Meshlet-Buffer Considerations

Meshlets are usually bound in the number of vertices and indices. For NVIDIA GPUs, those maximum values are recommended as 64 vertices and 126 primitives (378 indices for triangles) [Kub18].

This limitation on the vertex number allows the compression of the usually 32-bit indices into 8-bit indices. If vertex-reuse inside the meshlet is maximised this can save a substantial amount of data [Kub20].

Knowing those limitations a straightforward approach for a meshlet data structure is given by:

```
struct meshlet_fixed
{
    uint16_t meshIndex;
    uint8_t vertexCount;
    uint8_t triangleCount;
    uint32_t meshletVertices[64];
    uint8_t meshletIndices[380];
};
```

Here the `meshletIndices` array contains indices into the `meshletVertices` array. The `meshletVertices` array contains indices that point to the appropriate vertex data inside the vertex buffer. `meshletIndices` has a length of 380 elements such that we have a 4-byte-aligned struct. In total, this scheme enforces a fixed size of 640 bytes per meshlet.

Since we usually will not reach both the primitive and vertices bound we will end up with some unused data inside the vertex- and index-arrays. To mitigate this issue a two-buffer approach is proposed using a meshlet buffer with the following structure:

```
struct meshlet_redirect
{
    uint16_t meshIndex;
    uint8_t vertexCount;
    uint8_t triangleCount;
    uint32_t dataOffset;
};
```

It contains an offset into a 32-bit buffer. To avoid confusion with the *Index Buffer* present in a traditional pipeline, we will call this buffer the *Meshlet Index buffer*. It contains all the elements of the `meshletVertices` and `meshletIndices` arrays in sequential order. Each 32-bit unsigned integer packs four meshlet indices. Figure 3.3 visualizes this approach.

This solution is similar to a method with separate Vertex Index and Primitive Index Buffers as shown in [Kub20], except that those two buffers are combined into one *Meshlet Index* buffer. This saves approximately 32 bits per meshlet and may additionally reduce cache misses.

In the provided software, both implementations can be changed during runtime. They are compared in terms of buffer size and frame time in Table 3.2. We included the index buffer size to highlight the memory reduction compared to a traditional pipeline.

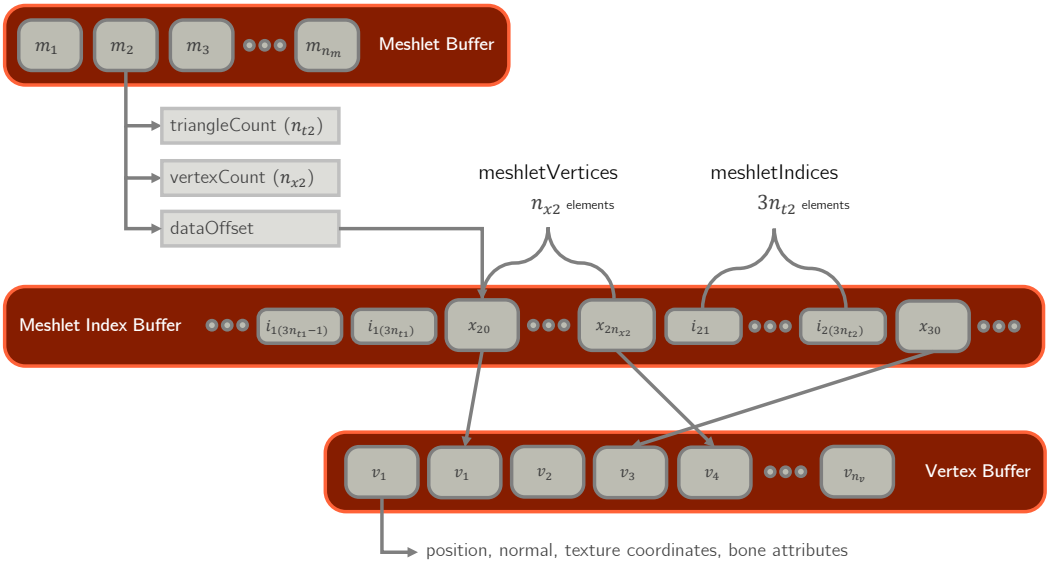


Figure 3.3: We propose to use one buffer for both the meshlet vertex indices x and the meshlet indices i. In this figure, n_m depicts the meshlet count and n_v the vertex count.

		Fixed		Redirect		
	Meshlets	**B.-Size**	**F.-Time**	**B.-Size**	**F.-Time**	**IB.-Size**
Playground	4589	2.80 MB	41.0 ms	**2.14 MB**	**40.1 ms**	3.93 MB
Mixamo Group	3478	2.12 MB	**35.3 ms**	**1.73 MB**	**35.3 ms**	3.41 MB
Monsters	4832	2.95 MB	56.4 ms	**2.38 MB**	**55.9 ms**	4.86 MB

Table 3.2: Comparison of two different meshlet buffer layouts. The model data was optimized for vertex reuse before running the tests. The frame times are taken with a copy count of 500 instances per model and no vertex compression was applied.

The results conclusively demonstrate that the two-buffer approach has significant benefits, using on average 25% less memory than the meshlet layout with a fixed meshlet size. When compared to the Index-Buffer used in a traditional pipeline, we observe approximately 95% storage savings. Given these results, we adopted the redirected approach for all other benchmarks unless explicitly stated otherwise.

3.3 Compression Codecs and Buffer Layouts

We conducted experiments to evaluate runtime performance and compression rates across seven different compression methods. We included two state-of-the-art methods for bone attribute compression and compared them to an uncompressed pipeline. Furthermore, we examined four variations of the proposed Difference Encoding. To ensure a fair comparison, we utilized the same code for bone animations and applied the same encoding/decoding

scheme for positions, texture coordinates, and normals as detailed in Chapter 2. For all algorithms, except the uncompressed codec, these values were stored with the following bit-depth inside the vertex-buffer:

```
3 x uint16 quantized vertex position
2 x uint16 octahedron encoded normal
2 x uint16 quantized texture coordinates
```

For the bone attributes, we ensure that all reference implementations, except the uncompressed codec, fit within 32-bit.

Uncompressed (UC)

For the uncompressed variant, 32-bit representations of all vertex components are used:

```
3 x float32 position
2 x float32 texture coordinate
3 x float32 normal
4 x uint32 bone indices
4 x float32 bone weights
```

This results in a fixed vertex-data size of *64 bytes per vertex*.

Optimal Simplex Sampling (OSS)

We use the OSS algorithm as proposed by Kuth et al. [KM21]. We compress the weights using OSS into 16-bit. The indices are stored inside a lookup table as detailed in 2.1.4. With 16-bit for the tuple index, we are well within the limits for our benchmark scenes (see Table 4.1).

To that end, we add the following entries to our vertex data structure:

```
1 x uint16 tuple index
1 x uint16 OSS encoded bone weights
```

This results in a fixed vertex-data size of *18 bytes per vertex*.

Permutation Coding (PC)

For this codec, we use the *Permutation Coding* algorithm as proposed by Peters et al. [PKM22]. They provide a Python script to calculate the best possible codec configuration. With a 16-bit budget for weights and indices, a general weight value count of 58 and an extra bit for w_2 is suggested. The weight value defines the maximum sample space for each component.

The algorithm returns one unsigned number which we attach to the vertex data:

```
1 x uint32 permutation encoded blend attributes
```

This results in a fixed vertex-data size of *18 bytes per vertex*.

Permutation Difference Codec (PDC8, PDC16, PDC32)

Our novel approach works as follows: As detailed in Section 2.2 we only store the difference for the bone attribute per vertex. The bit-depth r_m for those difference values Δb_{im} is variable and stored together with the reference value b_m on a meshlet basis.

The difference between the versions lies in the allowed values for the resolution r_m which can be set by modifying p in Equation 2.22:

- PDC8: $p = 8$ such that $r_m \in \{0, 8, 16, 24, 32\}$.
- PDC16: $p = 16$ such that $r_m \in \{0, 16, 32\}$.
- PDC32: $p = 32$ such that $r_m \in \{0, 32\}$.

Thus, we optionally attach to the vertex data:

```
1 x uint8/16/24/32 bone attribute difference
```

This results in a variable size of *14-18 bytes per vertex*. Additionally, we need a meshlet buffer extension adhering to the structure:

```
1 x uint32 bone attribute reference value
1 x uint32 resolution
```

This results in the requirement of an additional *8 bytes of storage per meshlet*.

32-bit Variant Optimisations

Since our whole bone attribute is packed inside 32-bit, this variant of our algorithm degrades to a binary differentiation between all vertices having the same bone attribute or not ($r_m \in \{0, 32\}$). Given this knowledge, we included the following optimizations for this codec:

- Saving the min is unnecessary as the 32-bit in the vertex buffer might as well just encode the bone attribute directly.
- In the case of $r_m = 0$, we use the 8 bytes in the extra meshlet buffer to not only store r_m, but also the weights and the tuple index in a format that can be decompressed faster than the permutation codec. Thus, we store the scaled weights (by equation 2.19) w'_2, w'_3, w'_4 alongside the tuple index. For $r_m = 32$ we set the tuple index to `0xFFFF` such that we don't have to store r_m explicitly. This leaves us with the following layout for the extended meshlet buffer:

  ```
  1 x uint_16 tuple index (0xFFFF when resolution is 32)
  3 x uint_16 scaled weights
  ```

Dual Permutation Difference Codec 16-bit (DPDC16)

To evaluate the use of the sorted and merged index table as proposed in Section 2.1.4, we implemented this codec based on the PDC16 with one modification: we pack the tuple index together with the weight permutation inside the payload for the permutation coding algorithm. Thus, we allocate:

- 13 bits for compressed weights,
- 5 bits for the permutation,
- 14 bits for the tuple index,

compressed in the blend attribute. Consequently, we must use a marginally worse codec for the permutation coding algorithm, with only 36 possible values, while still maintaining the extra count for w_2.

CHAPTER 4

Evaluation

In Section 4.1 we discuss the environment and the models we used for tests and benchmarks. To evaluate the effectiveness of the proposed method, we assess the potential reduction in memory consumption in Section 4.2. Furthermore, we compare the runtime performance of our method to two state-of-the-art solutions in Section 4.3. Regarding visual quality, our approach is by design comparable to the findings of Peters et al. [PKM22], as we utilize their proposed algorithm to calculate bone attributes.

4.1 Environment and Datasets

All benchmarks were taken on an NVIDIA GeForce RTX 3060 with a window resolution of 1920x1080. To measure the exact GPU timings we used GPU Timestamp Queries [1]. We averaged the frame-timings of 240 frames for each scenario. We froze the animation on the first frame of each respective scene and dropped all fragments to put more emphasis on the mesh stage. To achieve a higher workload, we rendered multiple instances of the same model. We do so by executing the pipeline multiple times.

The models used for benchmarking are listed in Table 4.1. We focused on models with varying vertex and face counts, ensuring that they also differ in the average number of bones influencing each vertex. We expect our methods to perform well on models with a low number of bones per vertex and a low number of unique index tuples. These conditions are typically satisfied with non-organic models, such as robots or cars.

[1] `https://docs.vulkan.org/samples/latest/samples/api/timestamp_queries/README.html`

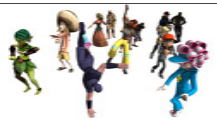
Screenshot	Name	Vertices	Faces	Bones	Meshlets meshoptimizer	Unique Tuples no compression	∅ bones per vertex
	Playground [2] [2]	9.69×10^5	3.44×10^5	210	15370	523	1.03
	Mixamo Group [2]	6.24×10^5	2.98×10^5	889	9839	4930	1.87
	Monsters [3]	1.27×10^6	4.24×10^5	3528	20468	18874	1.99
	Dragon [4]	1.14×10^5	3.80×10^4	146	1810	1476	2.23
	Michelle [2]	5.65×10^4	2.81×10^4	52	886	406	1.85
	Robot [5]	6.23×10^4	2.08×10^4	33	993	33	1.00

Table 4.1: Overview of all the scenes used for benchmarks, tests, and visualizations in this thesis.

[2]contains The Last Stronghold (animated) by Conrad Justin https://skfb.ly/oowEz
[2]contains animated and rigged Models from Mixamo by Adobe https://www.mixamo.com
[3]Ultimate Monsters Pack by quaternius https://skfb.ly/ozDYs
[4]Black Dragon with Idle Animation by 3DHaupt https://skfb.ly/FWLt
[5]Animated humanoid robot by pinguinoconpulgares https://skfb.ly/KE8x

4.2 Memory Consumption

To compare the overall memory consumption of our approach, we have to monitor:

- Vertex Buffer (*VB*): This buffer stores position, normal, texture coordinates, and blend attributes for each vertex. For UC, OSS, and PC, we have fixed buffer sizes that depend only on the vertex count. However, this varies for the PDC8, PDC16, PDC32, and DPDC16 approaches.
- Meshlet Buffer (*MB*): As outlined in Section 3.2, the Meshlet Buffer contains triangle and vertex data. The buffer size is constant across all evaluated algorithms.
- Additional Meshlet Buffer (*AMB*): Our difference codecs PDC8, PDC16, PDC32, and DPDC16 require additional information per meshlet, as detailed in section 3.3.
- Tuple Index Buffer (*TIB*): This buffer contains the contents of the lookup table, with each tuple stored as `4 x uint16`.

Table 4.2 contains all the relevant buffer sizes for our test scene, highlighting a substantial memory reduction with our proposed approach. PDC8 shows the best compression since we allow the most fine-grained bit-resolution for the bone-attributes. The stark contrast between the different buffer sizes is shown in Figure 4.1.

				Ours			
		UC	**OSS/PC**	**PDC8**	**PDC16**	**PDC32**	**DPDC16**
Playground	VB:	59.15 MB	16.63 MB	13.09 MB	13.14 MB	13.29 MB	13.13 MB
	AMB:	-	-	120.08 kB	120.08 kB	120.08 kB	120.08 kB
	TIB:	-	2.73 kB	2.73 kB	2.73 kB	2.73 kB	0.97 kB
	SUM:	59.15 MB	16.64 MB	**13.21 MB**	13.26 MB	13.41 MB	13.25 MB
Mixamo G	VB:	38.09 MB	10.71 MB	9.53 MB	9.88 MB	10.02 MB	9.93 MB
	AMB:	-	-	76.87 kB	76.87 kB	76.87 kB	76.87 kB
	TIB:	-	24.30 kB	24.31 kB	24.31 kB	24.31 kB	8.91 kB
	SUM:	38.09 MB	10.74 MB	**9.63 MB**	9.98 MB	10.12 MB	10.02 MB
Monsters	VB:	77.70 MB	21.85 MB	19.11 MB	19.73 MB	20.03 MB	19.79 MB
	AMB:	-	-	159.91 kB	159.91 kB	159.91 kB	159.91 kB
	TIB:	-	99.92 kB	99.92 kB	99.92 kB	99.92 kB	40.13 kB
	SUM:	77.70 MB	21.95 MB	**19.37 MB**	19.98 MB	20.29 MB	19.99 MB
Dragon	VB:	6.96 MB	1.96 MB	1.81 MB	1.88 MB	1.90 MB	1.88 MB
	AMB:	-	-	14.14 kB	14.14 kB	14.14 kB	14.14 kB
	TIB:	-	8.27 kB	8.27 kB	8.27 kB	8.27 kB	3.37 kB
	SUM:	6.96 MB	1.96 MB	**1.83 MB**	1.91 MB	1.93 MB	1.90 MB
Robot	VB:	3.80 MB	1.07 MB	0.87 MB	0.89 MB	0.90 MB	0.90 MB
	AMB:	-	-	7.76 kB	7.76 kB	7.76 kB	7.76 kB
	TIB:	-	0.26 kB	0.26 kB	0.26 kB	0.26 kB	0.07 kB
	SUM:	3.80 MB	1.07 MB	**0.88 MB**	0.89 MB	0.91 MB	0.91 MB

Table 4.2: This table summarizes all relevant buffer sizes for our test scenes. PDC8 compresses the vertex buffer the most. DPDC16 uses the smallest lookup table.

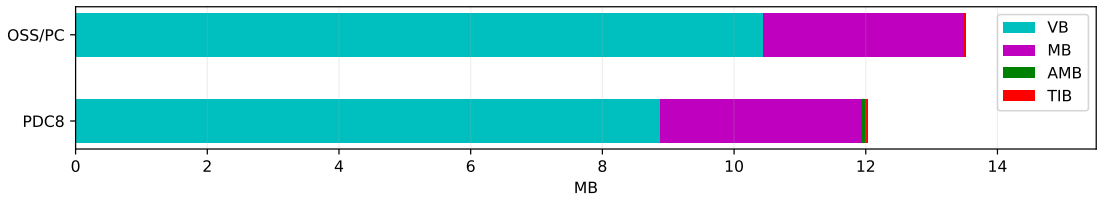


Figure 4.1: Visualization of the differences between buffer sizes on average showing the Vertex Buffer, Meshlet Buffer, and, specific to our bone attribute compression, the Additional Meshlet Buffer and Tuple Index Lookup Buffer.

4.2.1 Bone Attributes

If we extract the allocated size necessary for our bone attributes, we can calculate the average byte count per vertex for the blend attributes. We expect this value to be 32 bytes for the uncompressed version by design and approximately 4 bytes for the reference OSS and PC solutions, varying slightly due to the size of the index tuple lookup table. For a fair comparison, we must also include the AMB-buffer size for our Difference-Codecs. Table 4.3 and Figure 4.2 show the result of this metric.

As expected we get a substantial reduction for the blend attributes on scenes with mostly non-organic rigged meshes. In the `Playground` scene, a significant storage reduction of 92.75% is observed for PCD8 compared to state-of-the-art OSS/PC approaches. When compared to the uncompressed version, the storage reduction reaches 99.09%.

			Ours			
	UC	**OSS/PC**	**PDC8**	**PDC16**	**PDC32**	**DPDC16**
Playground	32.00	4.00	**0.29**	0.34	0.51	0.34
Mixamo G.	32.00	4.04	**2.18**	2.77	3.00	2.83
Monsters	32.00	4.08	**1.95**	2.46	2.71	2.46
Dragon	32.00	4.07	**2.81**	3.54	3.72	3.50
Robot	32.00	4.00	**0.83**	1.05	1.25	1.24
Average	32.00	4.04	**1.61**	2.03	2.24	2.08

Table 4.3: Average byte count per vertex including the additional memory necessary for the index tuple lookup table and the size of the additional meshlet buffer.

As the `Playground` scene might not be generally representative, we decided to focus on the average values of our findings. Those still show a reduction in the range of 45%-60% compared to OSS/PC and 93%-95% compared to the uncompressed codecs. The blend attributes with our scheme occupy only a small fraction of the overall vertex buffer, ranging from 11% for PDC8 to 16% for PDC32 (visualized in figure 4.3).

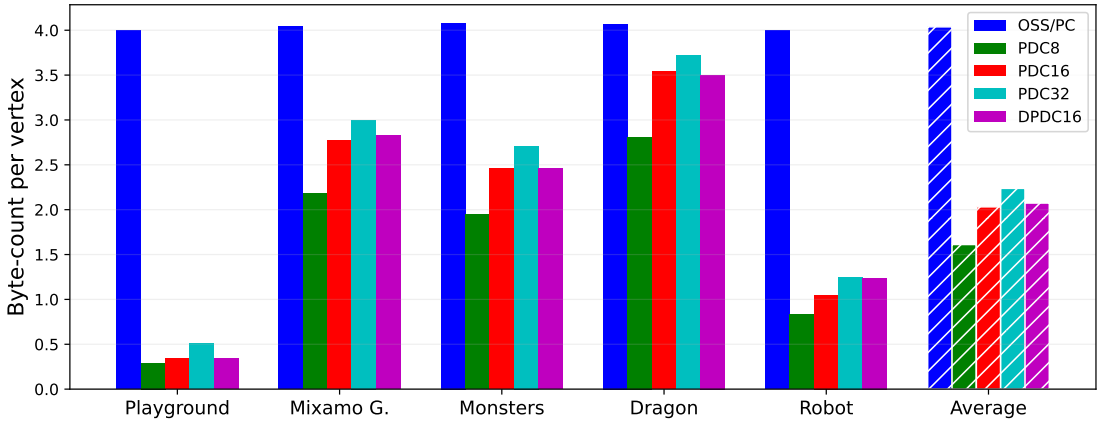


Figure 4.2: The average byte count per vertex for blend attributes is evaluated for the proposed encoding schemes. The Uncompressed (UC) scheme is deliberately excluded from the plot as its disproportionately large size would obscure the relative differences among the other values.

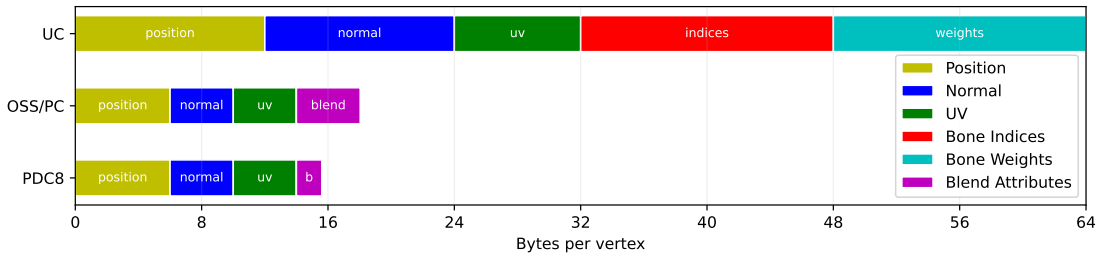


Figure 4.3: In our setup, the blend attributes constitute only 11% of the vertex buffer size (for PDC8), which is less than half of that required by current state-of-the-art algorithms. This figure already accounts for the additional storage needed for the extra meshlet and tuple index data.

4.3 Performance

As detailed in Section 4.1, we evaluated GPU frame times for all of our scenes. For our evaluations we duplicated the geometry multiple times: the larger scenes (`Playground`, `Mixamo Group`, `Monsters`) were rendered with 500 copies, while the smaller ones were rendered with 2000 copies. We averaged the timings over multiple frames. The results are presented in Table 4.4 and Figure 4.4.

Despite the substantial reduction of the blend attribute in some scenes (see Section 4.2), this does not significantly impact frame timing, as confirmed by our results. This is due to the fact that the blend attributes constitute a relatively small fraction of the entire vertex buffer layout. Therefore, our average memory gain of 60.24% for these attributes translates to only approximately 13.39% when considering the whole

vertex buffer. Additionally, decoding the compressed blend attributes introduces a minor overhead on the overall complexity.

The proposed algorithms exhibit slightly longer frame timings, as indicated by results that remain comparable to those of the permutation coding reference implementation. However, in the `Playground` scene, the algorithm outperforms the reference implementation in the PDC16 and PDC32 variants.

				Ours			
	UC	**OSS**	**PC**	**PDC8**	**PDC16**	**PDC32**	**DPDC16**
Playground	126.7	117.2	111.8	118.9	109.4	**108.9**	116.4
Mixamo G.	91.5	89.2	**87.7**	92.6	88.3	88.8	96.3
Monsters	207.2	**205.6**	205.7	217.2	208.4	209.2	227.5
Dragon	70.3	61.8	**60.4**	64.5	62.1	62.5	70.7
Robot	33.4	24.2	**22.6**	24.6	22.7	22.8	25.5
Michelle	31.4	23.8	**22.9**	25.0	23.1	23.5	26.6

Table 4.4: Performance comparison in *milliseconds [ms]*. `Playground`, `Mixamo Group` and `Monsters` were rendered with 500 copies each. `Dragon`, `Robot` and `Michelle` with 2000 copies.

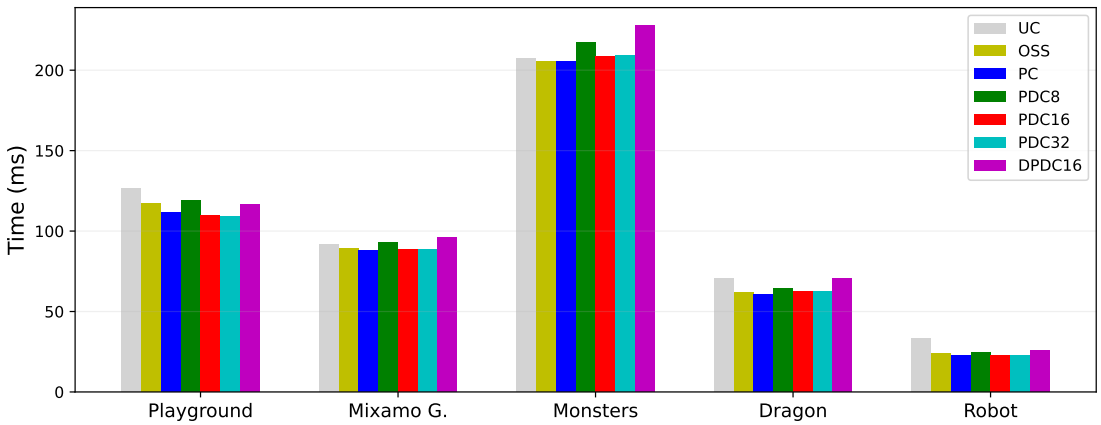


Figure 4.4: Frame timings for the tested encodings (see Table 4.4).

CHAPTER 5

Conclusion

In this thesis, we have explored methods for compressing vertex attributes, with a primary focus on blend attributes. We demonstrated that mesh shaders significantly enhance the compression of bone weights and bone indices by leveraging the local similarities of these attributes. The proposed algorithms are easy to integrate into a project already utilizing the mesh shading pipeline and achieve on average a 60% reduction in memory usage for the blend attributes compared to state-of-the-art methods while maintaining competitive performance.

Despite a reduction of the tuple lookup table buffer by 60.56%, our proposed algorithm for sorting and merging indices (see section 2.1.4) did not better the storage requirements or the frame times. Given the already minimal size of the table, the additional 5 bits required per vertex are disproportionate and do not justify this approach. As illustrated in Figure 4.1, the Tuple Index Buffer constitutes only a small fraction of the overall model data. Therefore, further compression of this buffer, even at the cost of a single additional bit-per-vertex, is unwarranted. Our proposed DPDC16 codec demonstrated worse performance in terms of both memory requirements and GPU frame time, while also decoding bone weights at a lower quality compared to the other proposed difference codecs.

Furthermore, our performance evaluation demonstrated that merely reducing the size of the vertex buffer offers limited potential for performance gains (see Section 4.3).

5.1 Future Work

We assume that the compression could be further optimized by utilizing a meshlet generation algorithm designed to maximize the similarities within the meshlets vertices. Although such an approach may not be advantageous for rendering purposes, it could be

valuable for compression, where even constraints on the number of vertices and indices per meshlet might not apply.

It may also be of interest to relax the sparseness constraint, allowing designers finer control over their animations. Permutation Coding scales effectively with more than four bones per vertex [PKM22]. Since our approach builds upon this algorithm, the same scalability applies to our scheme.

From a performance perspective, it could be beneficial to encode the weights and tuple index separately using a scheme with fewer performance implications (e.g., Equation 2.19). It is ultimately a trade-off between speed and storage requirements where our difference encoding scheme may help to keep the resulting size implications within reasonable bounds.

CHAPTER 6

Appendix

6.1 Proof $w_j < \frac{1}{j}$ for sorted weights

Proof by Contradiction

Assume that w_k is larger than $\frac{1}{k}$ for some $k \in [1, n]$ where n is the amount of weights per vertex:

$$w_k > \frac{1}{k}$$

Let's assume the weights are ordered such that:

$$w_j \geq w_{(j+1)} \quad \forall j \in [1, n-1]$$

Now, consider $w_k > \frac{1}{k}$.

$$w_1 \geq w_2 \geq \ldots \geq w_k > \frac{1}{k}$$

All w_j with $j < k$, must also be greater than $\frac{1}{k}$ because the weights are sorted. There are k such weights. The sum of these k weights need to be greater than $k\frac{1}{k} = 1$:

$$\sum_{z=1}^{k} w_z > 1$$

Since $k \leq n$ this contradicts the affinity constraint, which states that the sum of all weights must be equal to 1:

$$\sum_{k=1}^{n} w_j = 1$$

Since this contradiction arises from the assumption that $w_j > \frac{1}{j}$, the assumption must be false. Thus, for the affinity and sorting constraints to hold, $w_j \leq \frac{1}{j}$ must be true.

List of Figures

1.1 This figure illustrates the distinctions between the two geometry pipelines available in modern GPUs. Grey boxes represent fixed-function stages, while the red stages are fully customisable through the definition of appropriate shader programs [Kub18]. 2
1.2 In the traditional rendering pipeline, each vertex (red dots in left image) of the model is individually processed by the vertex shader. In a mesh shader-based pipeline, groups of vertices (right image) share a common storage structure and allow for efficient neighborhood access, which facilitates the use of more advanced compression algorithms. 2
1.3 The Mannequin character from the Mixamo Collection consists of 65 bones. Every vertex of the mesh is influenced by a subset of those bones with different magnitudes (bone weights). 3
1.4 The left images depict the scaled values of the first three bone weights (by equation 2.19) encoded as RGB colors. The right images show all unique bone index vectors in the `Michelle` and `Robot` model. 4

2.1 The octahedron encoding is defined by a mapping of the octants of a unit sphere onto the faces of an octahedron. The faces are mapped onto a unit square which can be stored in two components [CDE+14]. 8
2.2 Comparison of the distribution of unit vectors using different encoding methods. The naive quantized Euclidean method depicted here uses 5 bits per channel, resulting in a total of 15 bits. In contrast, methods (b) and (c) use 8 bits per channel resulting in a total of 16 bits. 9
2.3 Comparison of the amount and distribution of distinct tuple indices for different methods applied to the Mixamo `Michelle` character. We propose a sorting and merging step to reduce the number of unique tuples in the lookup table. The meshlets shown in [d] illustrate that most meshlets share the same tuple index. 10
2.4 Tuple-Reuse: Irrelevant indices are symbolized as ∞. The lookup table is then sorted, and only the last tuple of each corresponding group is saved [PKM22]. 11
2.5 Sorting and merging the bone indices allows for a better compression rate of the lookup table. 12

2.6 All possible combinations of bone weights lie inside a 3-dimensional simplex (a). A naive linear transformation can be applied to cover $\frac{1}{6}$ of all possible encodings (b). Peters et al. [PKM22] proposed a different linear transformation which is order-preserving (c). By shuffling the weights after applying Eq. 2.20 we can reach all possible values inside the unit cube. All possible permutations are shown in different colors (d). 14
2.7 Our Meshlet Difference Encoding algorithm stores bone attributes with varying bit resolutions inside the vertex buffer. The bit resolutions are visualized with different colors on the `Michelle`, `Robot` and `Dragon` models. The bone attributes are encoded with the Permutation Coding Algorithm [PKM22]. 17

3.1 The Skinned Meshlet Playground is a Vulkan application implemented specifically for this thesis. It allows us to compare our proposed algorithm to other state-of-the-art methods. 20
3.2 Comparison of a simple greedy meshlet generating approach (AVK-Default, left side) and an approach which takes vertex reuse, proximity and normal variance into account (Meshoptimizer, right). One random meshlet is highlighted in both images. 22
3.3 We propose to use one buffer for both the meshlet vertex indices x and the meshlet indices i. In this figure, n_m depicts the meshlet count and n_v the vertex count. 24

4.1 Visualization of the differences between buffer sizes on average showing the Vertex Buffer, Meshlet Buffer, and, specific to our bone attribute compression, the Additional Meshlet Buffer and Tuple Index Lookup Buffer. 32
4.2 The average byte count per vertex for blend attributes is evaluated for the proposed encoding schemes. The Uncompressed (UC) scheme is deliberately excluded from the plot as its disproportionately large size would obscure the relative differences among the other values. 33
4.3 In our setup, the blend attributes constitute only 11% of the vertex buffer size (for PDC8), which is less than half of that required by current state-of-the-art algorithms. This figure already accounts for the additional storage needed for the extra meshlet and tuple index data. 33
4.4 Frame timings for the tested encodings (see Table 4.4). 34

List of Tables

2.1 Scale-invariant quantization error for different bit depths 6
2.2 Errors for different normal encoding schemes on various datasets. The encoded data fits into 32-bit leaving 10-bit for the components of the Euclidean method and 16-bit for the components of the spherical and octahedral methods. . 9

3.1 Performance comparison between NVIDIA and Vulkan extensions for Mesh shading by monitoring *GPU frame timings*. NVIDIAs extension reaches peak performance on our hardware. Timings are taken with a copy count of 500 instances per model and no vertex compression. 21
3.2 Comparison of two different meshlet buffer layouts. The model data was optimized for vertex reuse before running the tests. The frame times are taken with a copy count of 500 instances per model and no vertex compression was applied. 24

4.1 Overview of all the scenes used for benchmarks, tests, and visualizations in this thesis. 30
4.2 This table summarizes all relevant buffer sizes for our test scenes. PDC8 compresses the vertex buffer the most. DPDC16 uses the smallest lookup table. 31
4.3 Average byte count per vertex including the additional memory necessary for the index tuple lookup table and the size of the additional meshlet buffer. 32
4.4 Performance comparison in *milliseconds [ms]*. `Playground`, `Mixamo Group` and `Monsters` were rendered with 500 copies each. `Dragon`, `Robot` and `Michelle` with 2000 copies. 34

List of Algorithms

2.1 Vertex Data Preparation for Meshlet Difference Encoding 16

Bibliography

[BBM24] Joshua Barczak, Carsten Benthin, and David McAllister. Dgf: A dense, hardware-friendly geometry format for lossily compressing meshlets with arbitrary topologies. In *Proceedings of the ACM SIGGRAPH Conference*, July 2024.

[CDE+14] Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT)*, 3(2):1–30, April 2014.

[CM02] Peter H. Chou and Teresa H. Meng. Vertex data compression through vector quantization. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):373–382, October 2002.

[Com24] Computer Graphics Group, TU Wien. Auto-vk-toolkit. `https://github.com/cg-tuwien/Auto-Vk-Toolkit`, 2024. Accessed: 2024-08-05.

[Eng20] Matthias Englert. Using mesh shaders for continuous level-of-detail terrain rendering. In *ACM SIGGRAPH 2020 Talks*, SIGGRAPH '20, New York, NY, USA, 2020. Association for Computing Machinery.

[JFB23] Mark Bo Jensen, Jeppe Revall Frisvad, and J. Andreas Bærentzen. Performance comparison of meshlet generation strategies. *Journal of Computer Graphics Techniques (JCGT)*, 12(2):1–27, December 2023.

[Kap17] Arseny Kapoulkine. meshoptimizer. `https://github.com/zeux/meshoptimizer`, 2017. Accessed: 2024-08-05.

[Khr24] Khronos Group. *SPIR-V Specification*, 2024.

[KM21] Bastian Kuth and Quirin Meyer. Vertex-blend attribute compression. In *Proceedings of the Conference on High-Performance Graphics*, HPG '21, pages 43–52, Goslar, Germany, 2021. Eurographics Association.

[KMWM21] Brian Karis, Jeremy Moore, Daniel Wright, and Martin Mittring. Nanite: A deep dive into the virtualized geometry system in unreal engine 5. In

ACM SIGGRAPH 2021 Advances in Real-Time Rendering, 2021. Accessed: 2024-08-09.

[KOK+23] Bastian Kuth, Max Oberberger, Felix Kawala, Sander Reitter, Sebastian Michel, Matthäus Chajdas, and Quirin Meyer. Towards practical meshlet compression, 2023.

[Kub18] Christoph Kubisch. Introduction to turing mesh shaders. `https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/`, 2018. Accessed: 2024-08-05.

[Kub20] Christoph Kubisch. Using mesh shaders for professional graphics. `https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/`, 2020. Accessed: 2024-08-05.

[LCL10] Jongseok Lee, Sungyul Choe, and Seungyong Lee. Mesh geometry compression for mobile graphics. In *Proceedings of the 2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5, USA, January 2010. IEEE.

[LD12] Binh Huy Le and Zhigang Deng. Smooth skinning decomposition with rigid bones. *ACM Transactions on Graphics*, 31(6), November 2012.

[MLDH15] Adrien Maglo, Guillaume Lavoué, Florent Dupont, and Céline Hudelot. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys*, 47(2), April 2015.

[MSS+10] Quirin Meyer, Jochen Süßmuth, Gerd Sussner, Marc Stamminger, and Günther Greiner. On floating-point normal vectors. *Computer Graphics Forum*, 29:1405–1409, June 2010.

[MSS24] Daniel Mlakar, Markus Steinberger, and Dieter Schmalstieg. End-to-end compressed meshlet rendering. *Computer Graphics Forum*, 43, January 2024.

[PBCK05] Budirijanto Purnomo, Jonathan Bilodeau, Jonathan Cohen, and Subodh Kumar. Hardware-compatible vertex compression using quantization and simplification. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 53–61, New York, NY, USA, 2005. Association for Computing Machinery.

[PKM22] Christoph Peters, Bastian Kuth, and Quirin Meyer. Permutation coding for vertex-blend attribute compression. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 5(1), May 2022.

[UKPW21] Johannes Unterguggenberger, Bernhard Kerbl, Jakob Pernsteiner, and Michael Wimmer. Conservative meshlet bounds for robust culling of skinned meshes. *Computer Graphics Forum*, 40(2):217–229, May 2021.