



HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg
Fakultät Elektrotechnik und Informatik

Studiengang: Informatik

Bachelorarbeit

Geometriedekompression von Dreiecksnetzen auf der GPU

Janek Foote

Abgabe des Arbeit: 13.03.2024

Betreut durch:

Quirin Meyer, Hochschule Coburg

Inhaltsverzeichnis

Abbildungsverzeichnis	4
Tabellenverzeichnis	5
Codebeispielverzeichnis	6
1 Einführung	7
1.1 Geschichtliche Ausarbeitung Thema Datenkompression	7
1.2 Steigende Komplexität	9
1.3 Ziel der Arbeit	10
2 Grundlagen	11
2.1 Brotli Kompressionsstandard	11
2.2 Parallele Datenverarbeitung	11
2.3 Die traditionelle Rendering Pipeline	14
2.3.1 Vertex Shader	14
2.3.2 Tessellation Stage	15
2.3.3 Geometry Shader	15
2.3.4 Pixel Shader	16
2.4 Compute Shader	16
2.5 Quantisierung von Gleitkommazahlen	17
2.6 Grundbegriffe der Datenkompression	17
3 Methodik	18
3.1 Ablauf des Experiments	18
3.2 Mesh Shader	19
4 Mesh Shader	20
4.1 Mesh Shading Pipeline	20
4.1.1 Mesh Shader	21
4.2 Meshlets	21
4.3 Implementierung eines Standard Mesh Shaders	22
4.4 Mesh Shader Implementation	24
4.5 Lokale Vertex Buffer	25
5 Kompressionsstandard Brotli-G	27
5.1 LZ77	27
5.1.1 Kodierung eines Codewortes	28
5.1.2 Dekodierung eines Codeworts	30

5.2	Huffman-Kodierung	31
5.2.1	Konstruktion einer Huffman-Kodierung	31
5.2.2	Resultate einer Huffman-Kodierung	32
5.3	GZIP	34
5.4	Brotli-G Compute Shader	34
5.5	CPU Ebene	34
6	Ergebnisse	35
6.1	Auswertung des Datensatzes	36
6.2	Auswertung der quantisierten Vertex-Daten	37
6.3	Auswertung eines großen Dreiecksnetzes	39
7	Fazit	42
7.1	Ausblick	42
	Literaturverzeichnis	43
	Ehrenwörtliche Erklärung	

Abbildungsverzeichnis

Abb. 1:	Youtube Statistik	8
Abb. 2:	SIMD Pattern	13
Abb. 3:	traditionelle Rendering Pipeline	14
Abb. 4:	Flussdiagramm Ablauf	19
Abb. 5:	Mesh Shading Pipeline	20
Abb. 6:	Lokaler Vertex Buffer	26
Abb. 7:	Huffman Code Beispiel	32
Abb. 8:	Der verwendete Datensatz	35
Abb. 9:	Fandisk Kompressionsergebnis	36
Abb. 10:	Welsh Dragon Kompressionsergebnis	37
Abb. 11:	Quantisiertes Stanford Bunny	38
Abb. 12:	Welsh Dragon Kompressionsergebnis quantisiert	39
Abb. 13:	David Kompressionsergebnis	40
Abb. 14:	David Kompressionsergebnis quantisiert	41

Tabellenverzeichnis

Tab. 1:	LZ77 Kodierungs Beispiel	29
Tab. 2:	Dekodierung mit dem LZ77-Algorithmus	30

Codebeispielverzeichnis

Code 1:	Standard Mesh Shader main-Methode	25
---------	---	----

1 Einführung

In der Computergrafik ist die Erzeugung eines Dreiecksnetzes eine gängige Methode zur Generierung von 3D-Modellen. Diese Modelle können in Topologie und Geometrie unterteilt werden. Für die Geometrie werden verschiedene Attribute benötigt. So werden die Positionen, die Normalenvektoren und Texturekoordinaten/Farbwerte für jeden Punkt des Dreiecksnetzes in single-precision floating point values (32 Bit Gleitkommazahlen) gespeichert. Für die korrekte Anordnung und Reihenfolge der Knotenpunkte ist die Topologie zuständig. Dabei ist die Datenkompression ein entscheidendes Thema. In einer Welt, in der digitale Daten schon lange ein wichtiges Thema sind, und dennoch immer weiter an Bedeutung gewinnen, ist die effiziente Speicherung und Übertragung ein wichtiger Gesichtspunkt. 3D Modelle werden so gut wie überall benötigt. Videospiele und Animationsserien wären ohne nicht vorstellbar. Architekten können ihre Ideen auch ohne Bleistift auf das Papier (oder den Bildschirm) bringen. Künstler wollen Modelle erschaffen, die den Eindruck gewinnen wollen, realitätsgetreu zu sein. Die Folge davon ist, dass diese Modelle stetig komplexer werden, und somit ein größerer Speicheraufwand benötigt wird. Um dem entgegenzuwirken, werden Methoden verwendet, diese digitalen Informationen zu komprimieren.

1.1 Geschichtliche Ausarbeitung Thema Datenkompression

Ursprünglich zur Repräsentation von Daten entwickelt, wurde der Morse Code zu einem der wichtigsten Werkzeuge für die Kommunikation des 19. Jahrhunderts. Bestehend aus zwei Grundbausteinen, einem kurzen und einem langen Signal, konnten einzelne Buchstaben kodiert werden. Erweitert man dieses Alphabet mit einem weiteren „Symbol“ einer Pause, die zwischen einzelnen Signalsequenzen eingelegt wird, können ganze Sätze übermittelt werden. Das bekannteste Werkzeug für den Morse Code ist der Telegraf, mit dem diese Signale über weite Strecken übertragen werden konnten. Die Erfindung des Morsecodes findet im 21. Jahrhundert nicht nur seinen Zweck in dramatischen Momenten des in Film und Fernsehens. Es war zeitgleich ein früher und großer Meilenstein und Wegbegleiter für die Kompression einer Datenquelle (in diesem Fall das Alphabet). Durch Untersuchungen einer großen Anzahl an Literatur kann eine Buchstabenhäufigkeit berechnet werden. Diese sagt aus, wie wahrscheinlich es ist, welcher Buchstabe in einem Text folgt, ohne den aktuellen Kontext, in Form von vorgehenden Buchstaben, zu betrachten. Da die Wahrscheinlichkeit eines Zeichens abhängig vom Alphabet ist, sollten diese nicht übergreifend verwendet werden. So sind die Buchstaben „E“ und „T“ die Buchstaben des englischen Alphabets, welche die höchste Auftrittswahrscheinlichkeit besitzen, während sich im deutschen Alphabet der Buchstabe „E“ von der Masse abhebt. Der Morse Code hat gezeigt, welchen Nutzen die Kompression von Information beinhaltet. Zu Kriegszeiten hat dieser eine effiziente und schnelle Übermittlung von Informationen ermöglicht. Dadurch konnte in

Krisenmomenten schnell reagiert werden, um so größeren Katastrophen frühzeitig abzuwenden, aber leider auch, solche zu verursachen.

Der Ist-Stand

Springen wir in die heutige Zeit sehen wir die Auswirkung von komprimierten Daten. Die meisten Menschen denken an JPEGs und PNGs, wenn sie an digitale Bilder denken. Bekanntere Videoformate sind MP4, AVI und FLV. Bei all diesen Formaten handelt es sich um komprimierte Rohdaten. Das Filesystem eines jeden relevanten Betriebssystems komprimiert beim Speichern von Daten diese automatisch. Zusätzlich dazu besteht noch die Möglichkeit, seine Daten manuell zu komprimieren mithilfe von Programmen wie 7Zip, WinRar oder WinZip. Datenkompression kann in so gut wie allen Bereichen angetroffen werden. Und die Gründe dafür sind simpel. Speicherplatz ist teuer, und das Ressourcenmanagement wird deutlich vereinfacht, wenn die benötigte Hardware minimiert wird. Betrachten wir das Streamen von Daten auf dem Beispiel des größten Videostreaming Dienstes Youtube. Laut Statistiken werden pro Minute hunderte Stunden an Videomaterial hochgeladen Tendenz steigend (Abb.). Um die Unmengen an Videos

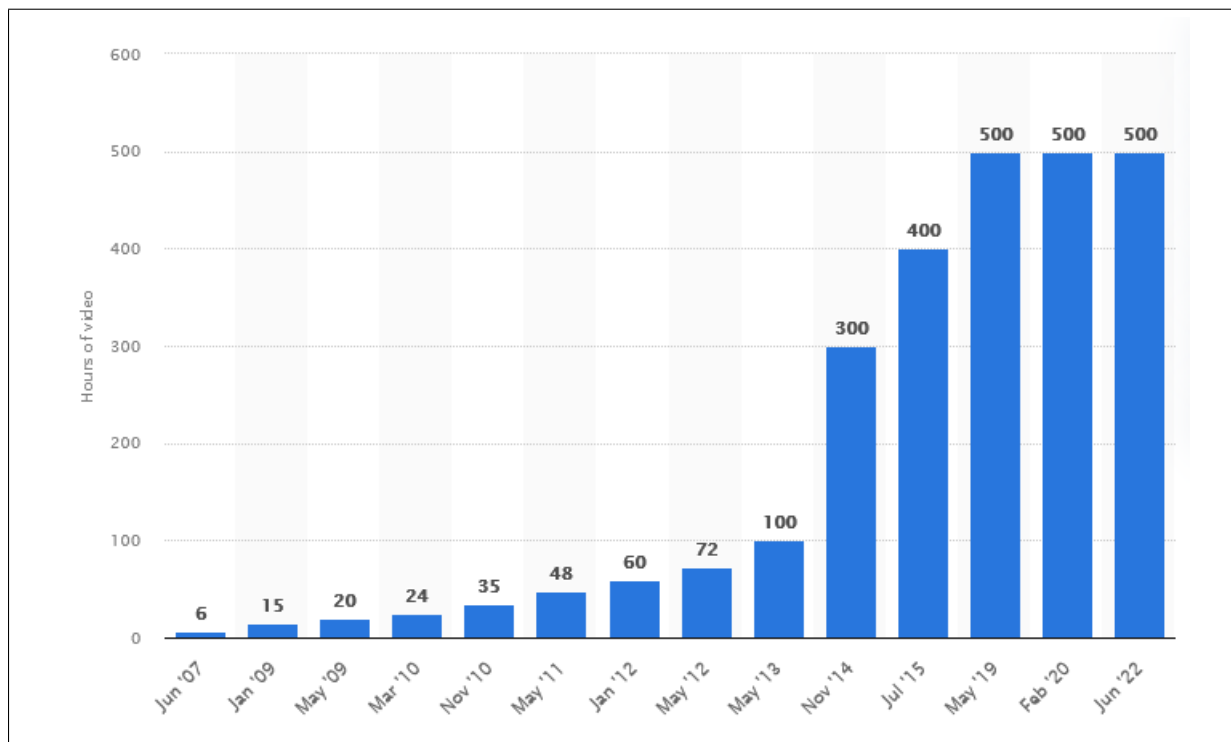


Abb. 1: **Anzahl der Youtube Videos** Die Anzahl an Minuten die auf Youtube hochgeladen werden. Abbildung von Statista:

<https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/>

zu speichern, benötigt Google riesige Serverfarmen, die auf dem gesamten Globus verstreut sind. Eine genaue Zahl ist der Öffentlichkeit nicht bekannt, es steht jedoch außer Frage, dass diese

nochmal um einiges höher ausfällt, würde es keine Verfahren zur Datenkompression geben.

Ein weiterer Gesichtspunkt ist der eigentliche Nutzen von Youtube, dem Streamen von Videos. Um ein Video sehen zu können, muss dieses von dem Youtube Server, zum Nutzer, dem Client übertragen werden. Durch die Komprimierung der Quelldateien sind die zu übertragenden Daten schon geschrumpft. Es können jedoch noch weitere Schritte absolviert werden, um die Daten für den Nutzer besser zugänglich zu machen. Dazu werden Verfahren wie Transcoding, Transsizing und Transrating verwendet. Transcoding beschreibt den Prozess, ein bereits komprimiertes Videoformat in ein anderes, eventuell für den Client besser zugeschnittenes Videoformat zu komprimieren. Das sollte jedoch nicht zu oft angewendet werden, da die Qualität beim wiederholten dekomprimieren und komprimieren verloren geht, sollten die Verfahren verlustbehaftet sein.

In vielen Fällen kann die originale Auflösung vom Endgerät nicht abgespielt werden, und wird deshalb von diesem auf eine niedrigere Auflösung skaliert. Beispielsweise wenn das Endgerät lediglich 1080p auflösen kann, aber ein Video in 4K Auflösung gestreamt werden soll. Trotzdem werden die vollen Daten des Videoformats empfangen. Um diese Verschwendung von Bandbreite zu sparen, wird Transsizing verwendet. Die originalen Daten werden in eine kleinere Auflösung skaliert, und anschließend übertragen.

Um die Bitrate zu minimieren wird Transrating verwendet. So kann die Auflösung beibehalten werden bei jedoch geringere Bitrate. Die Verfahren zur Minimierung des Datenstroms hören sich zunächst sehr mächtig an, sind jedoch mit Vorsicht zu genießen. Der Vorgang ist nämlich Verlustbehaftet und kann bei zu starker Nutzung zu Artefakten führen. Dafür ermöglicht es jedoch Menschen, deren Internetzugang ein Abspielen in hoher Qualität nicht zulässt, den Streaming Anbieter zu nutzen.

Neben dem Beispiel von Streaming Anbietern reihen sich noch viele weitere Beispiele, die einen riesigen Vorteil von der Datenkompression ziehen. In einem jedem BWL Grundlagenfach wird die Wichtigkeit des Wettbewerbsvorteils vermittelt. Ein Unternehmen muss auf Marktveränderungen schnellstmöglich erkennen. Insbesondere Unternehmen im Finanzbereich sind davon abhängig, damit die Datenanalyse schnellstmöglich auf Kursschwankungen reagieren kann.

1.2 Steigende Komplexität

Um die Realität bestmöglich darzustellen, werden Modelle stetig detailreicher, wodurch die Anforderungen an der Hardware steigen. In einer komplexen Szene können mehrere Millionen Dreiecke sichtbar sein, die je nach Anwendung, in Echtzeit gerendert werden müssen. Der Wunsch nach realistischeren Modellen in der Animationsfilm und Videospielbranche hat die Dreiecksanzahl von 3D Modellen in die Höhe schießen lassen.

1.3 Ziel der Arbeit

Wie man Anhand der Geschichte sieht, war die Datenkompression in ihrer frühen Zeit ein wichtiges Tool, um Informationen zu weiterzugeben. Dennoch hat und wird sie immer größere Bedeutung finden. Durch die rasch zunehmenden Digitalisierung wurden sowohl Massenmedien, die zur Speicherung von Daten dienen und verbesserten Leitungen bis hin zu Glasfaserkabeln, die eine höhere Bandbreite ermöglichen, entwickelt. Die Unterhaltungsbranche hat sehr von der Entwicklung profitiert. Aber nicht nur in der Bereitstellung ihres Streaming Services haben Kompressionsmethoden eine Relevanz bei Anbieter wie Youtube, Netflix oder Disney+.

Besonders in westlichen Ländern ist die Firma Walt Disney der wohl bekannteste Herausgeber von Animationsfilmen und -serien. In früher Zeit wurden Disney Produktionen von Hand gezeichnet. Da das ein sehr mühseliger und langwieriger Prozess ist, wurde eine neue Technik verwendet, die die Effizienz und Produktionsgeschwindigkeit maßgeblich erhöht. Anstatt die Szene von Frame zu Frame zu konstruieren, und so Charaktere und Objekte aus neuen Positionen und Blickwinkeln ständig neu zu zeichnen, wurden die zu Beginn angesprochenen 3D-Modelle entworfen. Das hatte zudem zur Folge, dass Effekte, die schwer zu zeichnen waren, realistischer in Simulationen zu berechnen waren, wie z.B. die Welleneffekte im Wasser. Weitere Anwendungsfälle für 3D-Modelle in der Unterhaltungsbranche sind Videospiele, visuelle Effekte in Film und Fernsehen und in *Virtual-Reality* (VR) und 3D-Hologramme bei Shows und in Themenparks. 3D-Modelle sind auch in anderen Bereichen anzutreffen. Architekten können ihre Vorstellung visualisieren, und so den Auftragsgeber ein erstes Bild zur Inneneinrichtung geben. 3D Drucker können diese Modelle mit Kunststoff herstellen. In der Fertigung werden Modelle mittel CNC Fräse robuster hergestellt.

Eine gängige Repräsentation von 3D-Modellen ist die eines Dreiecksnetzes. Damit die 3D-Modelle möglichst nah an dem ist, das sich der Künstler vorgestellt hat, können diese Dreiecksnetze eine große Anzahl an Dreiecken besitzen. In dieser Arbeit soll der neuartige Kodierungsstandard BrotliG getestet werden. Ein beliebiges Dreiecksnetz wird dafür zunächst in viele, kleine mini-Dreiecksnetze zerteilt. Die sogenannten *Meshlets* werden in einem eigenen Format gespeichert, das anschließend von BrotliG auf der CPU komprimiert wird. Die GPU bekommt die Daten, und dekomprimiert jedes einzelne Meshlet, um das gesamte Mesh abschließend zu Rendern.

2 Grundlagen

Obwohl Dreiecksnetze eine effektive Darstellung bieten, 3D-Modelle darzustellen, beanspruchen diese sehr viel Speicherplatz. Mithilfe von Brotli-G sollen diese auf der CPU komprimiert, und auf der GPU dekomprimiert werden, sodass diese fertig zur Darstellung sind, ohne viel Bandbreite zu nutzen. Damit der Weg von Komprimierung zu Darstellung verständlich ist, müssen einige grundlegende Dinge geklärt werden.

In diesem Grundlagenkapitel werden die von Brotli-G genutzten Algorithmen erläutert. Zusätzlich wird ein Ausblick auf die Grafikpipeline gegeben, und die Stellen betrachtet, bei denen weitere Verbesserungen vorgenommen werden können. Diese zeigt alle Transformationen, die die Daten eines Dreiecksnetzes von dem GPU Buffer bis zum Bildschirm durchläuft.

2.1 Brotli Kompressionsstandard

Brotli-G ist eine Weiterentwicklung des Brotli Kompressionsstand, der von AMD im Jahre 2022 entwickelt und veröffentlicht wurde. Die AMD Spezifikation bietet parallele Datenverarbeiten nach dem SIMD Prinzip (Kap. 2) auf Parallelrechnern, wie GPUs und Multithreaded CPUs. Zum Verständnis des von AMD veröffentlichten Kompressionsmodells ist zunächst ein Blick auf das Original erforderlich.

Brotli ist ein von Google Research entwickelter Kompressionsstandard, der 2013 veröffentlicht wurde. Er ist darauf ausgelegt, Webinhalte effizienter zu komprimieren als ältere Standards wie Gzip oder Deflate. BrotliG wurde mit bedacht auf Kompatibilität mit dem offiziellen Brotli entwickelt. So sollte Brotli auch in der Lage sein, Inhalte, die mit BrotliG komprimiert wurden, zu entschlüsseln. Zu beachten ist, dass dies nur in diese Richtung funktioniert, und somit Brotli das BrotliG Format nicht dekodieren kann [AMD22].

Brotli verwendet eine Kombination vieler Kompressionsalgorithmen, um Inhalte effizient zu komprimieren. Brotli's Kern besteht aus einem LZ77 Algorithmus, der in unterschiedlichen Ausführung auch in anderen Kompressionsstandard verwendet wird. Der LZ77 Algorithmus wird zusätzlich mittels Huffman Codierung optimiert.

2.2 Parallele Datenverarbeitung

Michael Flynn unterteilte Rechnerarchitekturen in Kategorien, die Abhängig von der Anzahl der Instruktions- und Datenströme sind.

Die Instruktions- und Datenströme:

SI (**S**ingle **I**nstruction)

MI (**M**ultiple **I**nstruction)

SD (**S**ingle **D**ata)

MD (**M**ultiple **D**ata)

können kombiniert werden.

Dadurch ergeben sich die vier Rechnerarchitekturen *SISD*, *SIMD*, *MISD*, *MIMD*.

SISD (Single Instruction, Single Data)

Die am häufigsten anzutreffende Rechnerarchitektur. Bekannter unter den Namen Von-Neumann Architektur bearbeitet diese Architektur die auf dem Speicher befindlichen Daten seriell. Man redet auch von skalaren Operationen auf die Daten. Rechnerarchitekturen mit SISD sind leicht zu verstehen und die Verarbeitung ist vorhersehbar. Der Preis dafür ist jedoch die langsame Geschwindigkeit gegenüber parallelen Architekturen.

SIMD (Single Instruction, Multiple Data)

Um die Geschwindigkeit zu erhöhen, werden Daten, auf denen die selbe Operation ausgeübt wird, parallel verarbeitet. Das ist bei der Berechnung von Vektoren und Matrizen von Vorteil. Betrachten wir die Addition zweier Vektoren, so kann der resultierende Vektor berechnet werden, wenn die einzelnen Komponenten der Vektoren addiert werden (siehe Abb. 2). Der Vertex Shader macht von diesem Konzept Gebrauch, während dieser seine per-Vertex Operationen ausführt [DC96].

MISD (Multiple Instruction, Single Data)

Um alle Kombinationen von Daten und Instruktionsströmen zu zeigen wurde auch MISD definiert. Die Rechnerarchitektur bezieht sich darauf, dass auf nur einem Datenpunkt verschiedene Operationen ausgeführt werden. Für eine lange Zeit war diese Art von Rechnerarchitektur rein theoretisch anzutreffen, da weder

MIMD (Multiple Instruction, Multiple Data)

Wie auch die SIMD Architektur ist MIMD in Parallelrechnern anzutreffen. Das Operationsprinzip von MIMD ist die Datenparallelität. Das Funktionsprinzip von MIMD-Rechnern umfasst die

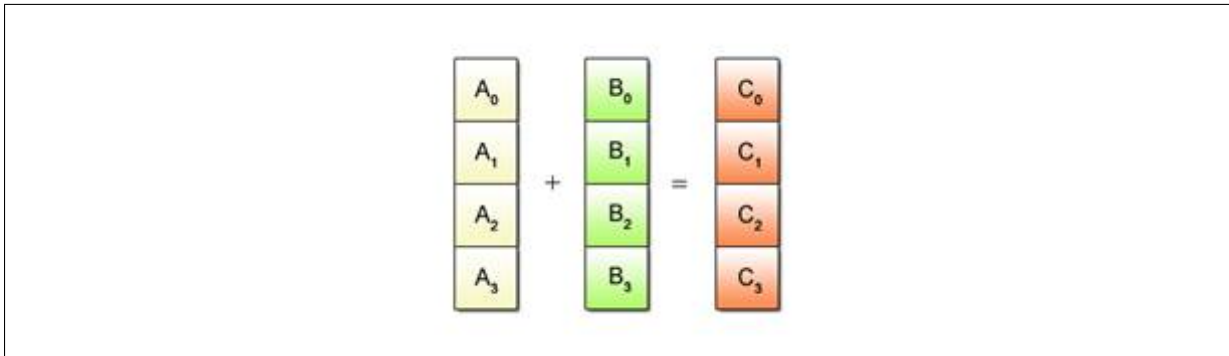


Abb. 2: **Parallele Addition von Daten** In der Abbildung ist eine Addition verschiedener Daten zu sehen. Da die selbe Operation ausgeübt wird können die einzelnen Komponenten parallel verarbeitet werden. Abbildung ist aus <http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsofSIMDProgramming.html>

gleichzeitige Ausführung von Anweisungen durch mehrere Prozessoren, die entweder über gemeinsame Variablen oder durch Nachrichten miteinander kommunizieren [DC96]. TODO später [JBG17]

2.3 Die traditionelle Rendering Pipeline

Um den Nutzen der neu vorgestellten Task- und Mesh-Shader Pipeline zu verstehen, muss zunächst die traditionelle Pipeline dort betrachtet werden, wo sie verbessert werden kann. Die Rendering Pipeline besteht aus eine Reihe von programmierbaren (Abb. 3 grün dargestellt) und fixed-function (Abb. 3 türkis dargestellt) Stages. Dazu kommt, dass einige dieser Stages optional sind.

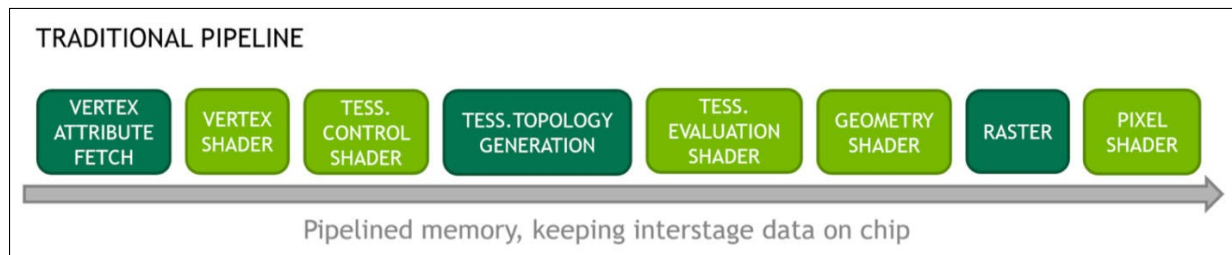


Abb. 3: **traditionelle Rendering Pipeline**

Die Abbildung beschreibt den Verlauf durch die einzelnen Shader Stages, die jeder Vertex macht. Entnommen wurde diese aus dem NVidia Blogpost [Kub18]

Im GPU Memory angekommen, liest die *Vertex Attribute Fetch* Stage die Vertex Daten aus, und sendet diese an den Vertex Shader. Die Vertex Daten werden dort in die benötigten Koordinatensysteme transformiert und der optionalen Tessellation Stage weitergegeben, falls diese verwendet wird. Die Tessellation Stage ist dazu da, Patches von Primitiven in kleinere Primitiven zu unterteilen. Der optionale Geometry Shader kann dazu verwendet werden, weitere Vertices zu generieren. Im Rasterizer angekommen, werden Primitiven verarbeitet und daraus Fragmente berechnet, denen der Fragment Shader zum Abschluss ihre Farbe gibt.

Im Folgenden werden die einzelnen Stages nochmal genauer erläutert.

2.3.1 Vertex Shader

Zunächst wird der vom Entwickler programmierbare Vertex Shader angesteuert. Dieser ist nicht optional, da alles nachfolgende auf den ausgegebenen Vertices aufbaut. Hier können Operationen auf einzelnen Vertices ausgeführt werden. Dafür wird der Vertex Shader für jeden Vertex einzeln aufgerufen. Hier zeichnet sich das SIMD Modell der GPU aus (Kap 2.2), da der Vertex Shader von mehreren Prozessoren auf unterschiedlichen Vertices zeitgleich operiert. Die Inputs des Vertex Shaders werden mittels *Vertex Attribute Locations* in den Shader eingebunden. Der Shader kann dadurch die Positionen, Normalen und Texturkoordinaten von der CPU aufnehmen. Eine Einschränkung, die dabei aufkommt, ist das Verhältnis von Eingabe und Ausgabe Vertices. Der Shader erwartet, dass für jeden Eingabe Vertex auch ein Vertex ausgegeben wird. Die Vertex Position wird für gewöhnlich in den Clip-Space transformiert, und der Pipeline weitergegeben.

2.3.2 Tessellation Stage

Die Ausgabe Vertices des Vertex Shaders gelangen anschließend in die optionale Tessellation Stage. Der generelle use-case ist, einen Patch an Primitiven in wiederum kleinere Primitiven zu verarbeiten. Die Tessellation Stage wird in drei Schritte unterteilt. Darunter ist mit dem *Tessellation Control Shader* (TCS) ein optional programmierbarer Schritt, eine fixed-function mit der *Primitive Generation* und einen programmierbaren *Tessellation Evaluation Shader* (TES).

Tessellation Control Shader (TCS)

Der *Tessellation Control Shader* (TCS) (der wiederum optional ist), ist ein geeigneter Schritt um das *LOD* (Level of Detail) zu berechnen und unter gewissen Voraussetzungen vorab einige Patches zu cullen. Ein Patch beschreibt eine Anzahl an Primitiven. Aus der Subdivision dieses Patches werden weitere Vertices berechnet, die zur Verarbeitung in den nächsten Schritt der Pipeline geschickt werden. Im TCS wird der Grad der Tessellation, das Spacing zwischen subdivided Punkten und die gewünschte Topologie festgelegt. Genauer gesagt wird hier gesetzt, wie oft die Primitiven unterteilt werden und welche Form diese am Ende haben sollen (triangle, quad, isolines).

Tessellation Topology Generation (TPG)

Mit der fixed-function stage des Tessellation Schritts werden die Primitiven mittels den im TCS bestimmten Parametern unterteilt. Die Koordinaten werden anschließend für den Tessellation Evaluation Shader berechnet. Diese unterteilt die Patches abhängig von den Berechnungen der TCS.

Tessellation Evaluation Shader (TES)

Der *Tessellation Evaluation Shader* hat den einfachsten Job und realisiert lediglich die Arbeit, die von den zwei vorherigen Stages verrichtet wurde. Die berechneten Koordinaten des TPG werden in dieser Shader Stage interpoliert, um die neuen Vertices zu generieren. Abschließend werden die aus der Subdivision berechneten Vertices ausgegeben. Wenn der optionale TCS nicht genutzt wird, werden default Parameter für den TPG benutzt. [CR12][Car22]

2.3.3 Geometry Shader

Ein weiterer optionaler Schritt in der traditionellen Grafikpipeline ist der *Geometry Shader*. Er bekommt eine Primitive als Input, und kann keine oder auch mehr Primitiven ausgeben, als er

bekommen hat. Die Fähigkeit zusätzliche Vertices zu generieren ist auch das, was den Geometry Shader besonders macht. Der Geometry Shader bekommt seinen Input entweder vom TES, oder, wenn die Tessellation Stage keine Verwendung findet, vom Vertex Shader und leitet seine Ausgabe an den Fragment Shader weiter. Um Bandbreite zwischen CPU und GPU zu sparen, kann ein Geometry Shader ein Dreiecksnetz mit wenigen Dreiecken erweitern und dieses so detaillierter gestalten. Ähnlich wie bei der Tessellation, die auf *Patches* von Primitiven agiert, verarbeitet der Geometry Shader die Primitiven an sich.

2.3.4 Pixel Shader

Der Pixel bzw. Fragment Shader ist der letzte programmierbare Schritt der Grafikpipeline. In diesem werden die transformierten Vertices und Primitiven schlussendlich gezeichnet. Der Pixel Shader operiert jedoch nicht auf diesen Daten, sondern auf sogenannten *Fragmenen*. Das bedeutet, bevor der Pixel Shader seinen Input bekommt, müssen Vertices und Primitiven erst durch den Rasterizer. Nun liegt es am Entwickler, den einzelnen Fragmenten ihre Farbe zu geben. In einem Modell werden per-Vertex Texturkoordinaten gesetzt, die auf eine Texturemap verweisen. Im Fragment Shader wird diese Texturemap mittels Sampler interpoliert. Zusätzlich müssen noch Materialeigenschaften beachtet werden. Alternativ kann jeder Vertex auch seinen eigenen Farbwert besitzen.

Um der Szene mehr Realismus beizusteuern, kann im Fragment Shader auch ein Lichtmodell implementiert werden. Beispiele dafür sind *Flat shading*, *Gouraud shading* und *Phong shading*. Für die Lichtberechnung werden die Oberflächen-Normalen benötigt. Diese sind entweder in einem Dreiecksnetz gegeben, oder müssen noch berechnet werden. Ausgabe des Pixel Shaders ist wiederum ein Fragment.

2.4 Compute Shader

Der Compute Shader gehört nicht zur traditionellen Grafikpipeline, kann darin aber seinen Nutzen finden. Sie dienen dazu, jede Mögliche Information die gewünscht ist auf der GPU zu berechnen. Anders als bei den Shader Stages der traditionellen Grafikpipeline (Kap. 3), erwartet der Compute Shader keine definierten Input/Output Daten, wie beispielsweise der Vertex Shader, der als Input und Output einen Vertex erwartet. Der Compute Shader kann also willkürliche Daten verarbeiten und dabei noch die Parallelisierung der GPU nutzen [Ope24].

Wie schon gesagt erhält der Compute Shader keine Input Variablen wie beispielsweise der Vertex Shader. Im Gegensatz zu diesem werden benötigte Daten mittels Buffer und „Shader Ressource

Views“ auf die GPU geladen (in D3D12). Aber ganz ohne Inputs kommt der Compute Shader nicht aus. Vor Aufruf des Compute Shaders muss bestimmt werden, mit wie vielen Threads dieser arbeiten soll. Der Aufruf der Dispatch Methode mittel Grafik API führt dazu, das der aktuell aktive Compute Shader aufgerufen wird. Die Dispatch Methode nimmt die Anzahl an Threads in drei Dimensionen als Argument.

Dafür gelten jedoch Hardware Limitierungen. Für die Anzahl der Threads muss gelten

$$\begin{aligned} numThreadsX, numThreadsY, numThreadsZ &\leq 128 \\ numThreadsX * numThreadsY * numThreadsZ &= 1024 \end{aligned}$$

(Für Compute Shader Version 5_0)

Um das SIMD Konzept des Compute Shaders zu verstehen sind zwei Variablen elementar wichtig. SVGroupThreadID und SVGroupID. TODO

2.5 Quantisierung von Gleitkommazahlen

2.6 Grundbegriffe der Datenkompression

Entscheidungsgehalt Redundanz Entropie

3 Methodik

Die vorliegende Bachelorarbeit beschäftigt sich mit der Dekodierung verschiedener Dreiecksnetze mittels dem Kodierungsstandard Brotli-G. Diese Methodiksektion dient dazu, einen detaillierten Einblick auf die Durchführung und Analyse des Experiments zu geben. Das Experiment zielt darauf ab, komprimierte Dreiecksnetze auf der GPU zu dekomprimieren. Insbesondere sollen das Kompressionsverhältnis, Dekompressionsgeschwindigkeit und die visuelle Qualität quantitativ ausgewertet werden.

3.1 Ablauf des Experiments

In diesem Abschnitt folgt eine kleine Beschreibung, wie die Kompressionspipeline aussieht. In den Folgenden Kapiteln werden die einzelnen Teilschritte genauer erläutert.

Zu Beginn muss der Datensatz mittels Brotli-G kodiert werden. Der Einfachheit halber wird in diesem Abschnitt von einem einzigen Dreiecksnetz gesprochen. Der Meshoptimizer von Zeux [Zeu] ist dafür verantwortlich, aus den Positionen und Indizes die Meshletdaten zu generieren. Dazu wurde ein Binärformat entworfen, welches die relevanten Daten zum Darstellen des gesamten Dreiecksnetzes speichert. Das Binärformat besteht dementsprechend aus dem Meshlet Descriptor, Vertex Ressourcen (Positionen und Normalen) und den Indizes zur Primitivengenerierung.

Dieses Binärformat wird als gesamtes komprimiert. Anschließend werden die GPU Ressourcen für die Eingabe (komprimiertes Dreiecksnetz) und Ausgabe (dekomprimiertes Dreiecksnetz) angelegt.

Für die Ausgabe wird eine „Unordered Access View (UAV)“ verwendet. Wie der Name schon vermuten lässt, bietet diese eine flexiblere Möglichkeit, gleichzeitig an verschiedenen Orten zu lesen und schreiben. Besonders von Vorteil ist dieser Ressourcentyp für die parallele Verarbeitung. So können einzelne Threads von der Ressource lesen/schreiben, ohne warten zu müssen, bis ein anderer Thread die Ressource wieder freigibt [Mic21].

Die UAV wird im Compute Shader als Output Buffer gesetzt, und mit den dekomprimierten Daten des Dreiecksnetzes gefüllt.

Abschließend wird die UAV im Mesh Shader gesetzt und die Meshlets und somit das gesamte Dreiecksnetz werden aus den Binärdaten rekonstruiert.

Der gesamte Vorgang ist in Abbildung 4 zu sehen.

Ist dieser Schritt abgeschlossen, könnten die Daten der UAV auf der CPU ausgelesen werden, die Buffer der Mesh Shader verwendet gefüllt werden und in den GPU RAM geschrieben werden. Dieser Schritt ist jedoch als unnötig anzusehen, wenn nicht noch zusätzliche Informationen mit dem dekomprimierten Dreiecksnetz berechnet werden müssen. Der Output Buffer des

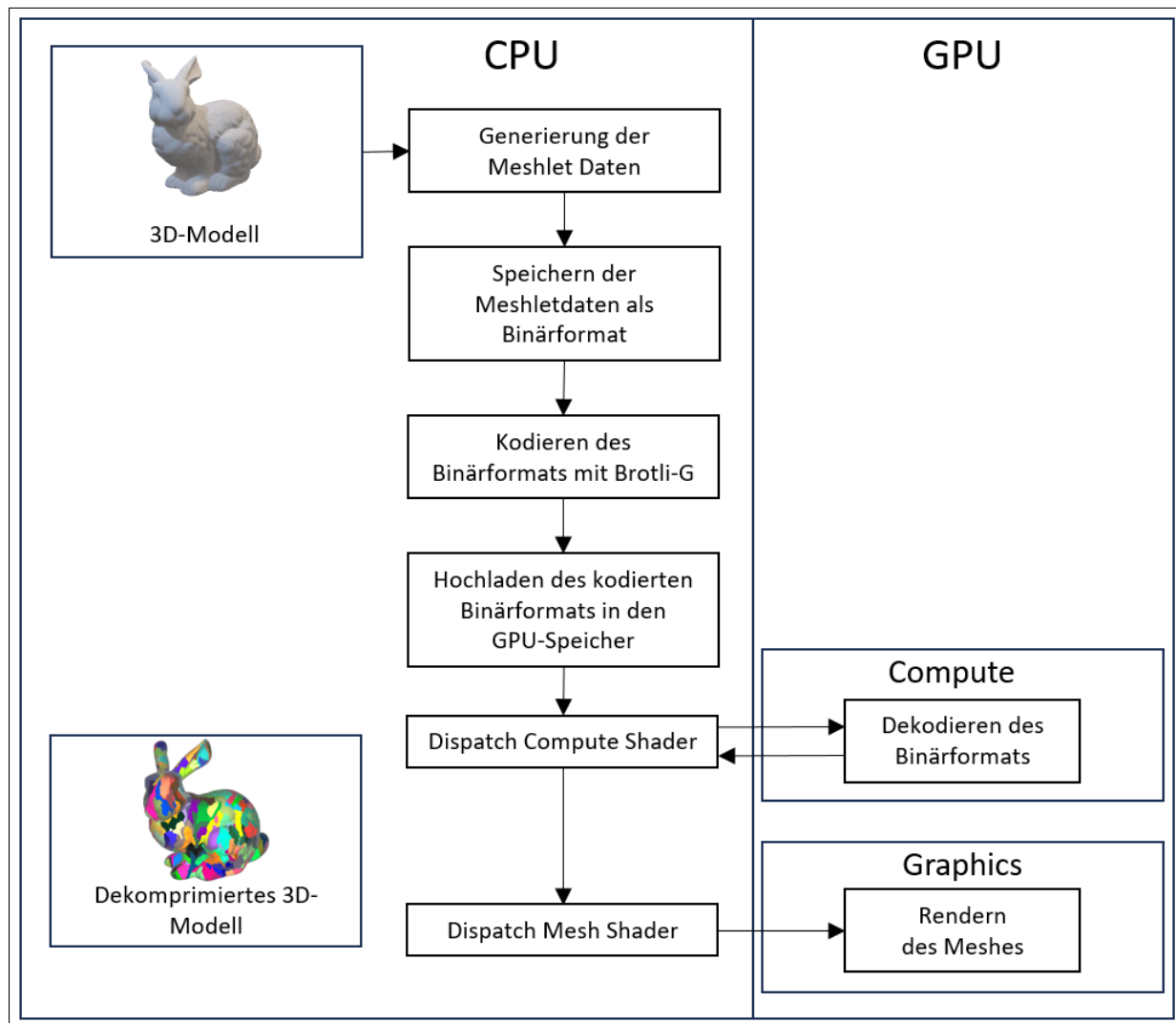


Abb. 4: **Flussdiagramm Ablauf** Abbildung der Dekompressionspipeline.

Brotli-G Dekodierers beinhaltet schon die benötigten Meshlet Daten, um das Dreiecksnetz zu rekonstruieren. So kann ein GPU Buffer außerhalb von Brotli-G angelegt werden, den Brotli-G als Output Buffer verwendet. Dadurch wird dieser Buffer nicht freigegeben, nachdem der Dekompressionsschritt von Brotli-G abgeschlossen ist. Abschließend übergibt man dem Mesh Shader den Output Daten Buffer und kann aus diesem das Dreiecksnetz rekonstruieren.

3.2 Mesh Shader

4 Mesh Shader

Die Architektur, auf der die neuartigen RTX-GPUs von Nvidia aufbauen, erweitert die Möglichkeiten, wie die Parallelisierung von GPUs genutzt werden kann. Mit der GeForce RTX 20er Serie wurden die ersten GPUs mit der Turing Architektur veröffentlicht, die sich auch an Privatpersonen richtet. Als großer Verkaufspunkt wurde bereits früh mit den Möglichkeiten von Real-time Raytracing und Deep Learning durch Tensor Core geworben [Bur20]. Eine wesentliche Änderung an der Grafikkipeline wird jedoch bis heute noch relativ wenig Beachtung geschenkt. Mit dem Shader Model 6 hat NVidia ihre sogenannte „next-generation shading Pipeline“ vorgestellt. Damit wird eine Alternative zur traditionellen Shading Pipeline gestellt, die dem Entwickler mehr Freiheit überlässt, die Parallelisierbarkeit der GPU zu nutzen. Der Mesh Shader hat die Eigenschaften des Compute Shaders (Kap. 2.4), der Daten auf der GPU parallel verarbeiten kann. Auch Geometrie Daten können mithilfe des Compute Shaders berechnet werden, jedoch ist der Compute Shader kein Teil der traditionellen Grafikkipeline und findet dadurch seinen Nutzen auch außerhalb des Renderings [Ile22]. Mit der *Mesh Shading Pipeline* wurde die Möglichkeit der Parallelisierung des Compute Shaders mit der neuen Rendering Pipeline verknüpft. Anders als bei der herkömmlichen Grafikkipeline erhält der Mesh Shader seine Daten direkt vom Speicher. Dadurch öffnen sich Türen für den Entwickler, da er komprimierte Daten direkt in den GPU Speicher laden kann, um die Daten dann effizienter auf dieser zu dekomprimieren.

4.1 Mesh Shading Pipeline

Die Mesh Shading Pipeline wirft einige Shader Stages der traditionellen Grafikkipeline aus Kap. 2.3 raus, und bietet die Funktionalitäten in den neuen Task- und Mesh Shadern an.

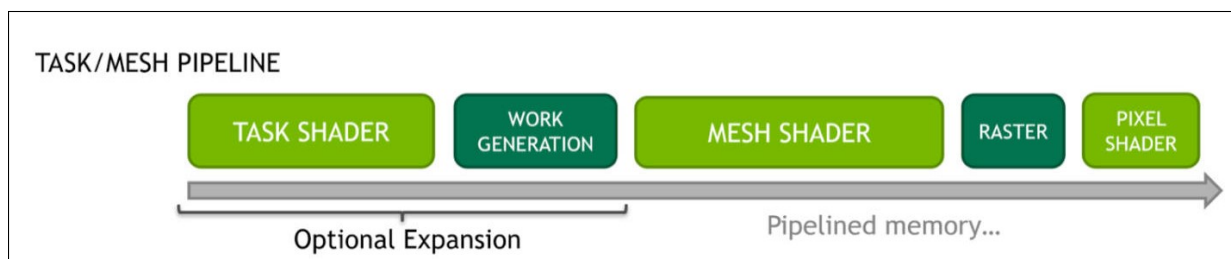


Abb. 5: **Mesh Shading Pipeline** Abbildung der Mesh Shading Pipeline. Die Abbildung ist aus dem NVidia Blogpost [Kub18]

Wie in der Abbildung 10 zu sehen ist, durchlaufen die Vertex Daten in der Mesh Shading Pipeline zunächst den Task Shader. Ähnlich wie bei Compute Shadern (Kap. 2.4), werden hier die Anzahl der workgroups an den Mesh Shader versendet. Der Mesh Shader arbeitet auf Threads. Die Ein- und Ausgabe des Mesh Shaders werden vom Entwickler festgelegt. So können wie beim Compute Shader auch arbeiten verrichtet werden, die nicht direkt fürs Rendering wichtig

sind. In dieser Arbeit wird der Mesh Shader zum dekomprimieren des Dreiecksnetzes genutzt. Zum Rendern müssen jedoch wieder Fragments ausgegeben werden, die in den unveränderte Rasterization und Pixel Shader Stages verarbeitet werden. [Kub18]

4.1.1 Mesh Shader

Da die optionale Task Shader Stage in dieser Arbeit nicht verwendet wird, betrachten wir die Pipeline aus dem Szenario, das die Anzahl an Workgroups des DirectX Dispatch direkt dem Mesh Shader gegeben wird. Die Nummer an Workgroups beschreibt, wie viele Kerne verwendet werden sollen. Bei Mesh Shadern bietet es sich an, die Anzahl der Workgroups auf die Anzahl an Meshlets zu setzen. Jede Workgroup hat außerdem eine Anzahl an Threads. Die Funktionsweise der Threads wurde bereits im Grundlagenkapitel 2.4 erläutert. [QUELLE!!!] Eine Workgroup besteht aus Threads, die mit *Shared Memory* arbeiten.

4.2 Meshlets

Um die neuartigen Shader für das Rendering zu verwenden, wird empfohlen, das gesamte Mesh in kleinere Subsets, sogenannte Meshlets, zu unterteilen. Die traditionelle Grafikpipeline verarbeitet die Daten des Dreiecksnetzes in serieller Manier. Dadurch kommt es jedoch zu Bottlenecks. In der traditionellen Pipeline werden Vertex und Primitiven vor der Vertex Shader Stage zugeschnitten und in kleine Clustern verarbeitet. Dazu wird der Primitive Distributor vor der Vertex Shader Stage aufgerufen. Dieser liest die Daten des Index Buffers und generiert dementsprechend möglichst performant diese Cluster an Daten. Der Schritt des Primitive Distributor ist jedoch eine fixed-function Stage der Grafikpipeline, wodurch der Entwickler keinen direkten Zugriff hat. Das hat zur Folge, dass die Cluster nicht auf die Bedürfnisse des Entwicklers und dessen Implementierung angepasst werden können. Zuzüglich werden die Cluster zu jedem Frame, bzw. vor jedem Aufruf der Pipeline neu generiert. Dieser Schritt ist redundant, sollte das Dreiecksnetz zur Laufzeit unverändert bleiben [Car22], [Kub18].

Durch die Compute Shading Natur des Mesh Shaders ist der Input der Daten nicht mehr festgelegt wie bei der traditionellen Pipeline. Dadurch kann der Entwickler seine eigenen Implementierungen zur Generierung von Meshlets verwenden. Anders als bei der herkömmlichen Grafikpipeline werden Meshlets auf CPU Ebene erstellt. Dazu werden Vertex Positionen und Indizes benötigt. Die Anzahl der Vertices und Primitiven muss im Vorfeld festgelegt werden. Die Auswahl der Meshletgröße ist abhängig von der verwendeten GPU. So wird im NVidia Blogpost „Introduction to Turing Mesh Shaders“ eine maximale Anzahl an Vertices von 64, und Primitiven von 126 empfohlen. Es werden 126 statt 128 Primitiven empfohlen, da 4 Byte für die

Anzahl der Primitiven verwendet werden, die im selben Block Speicher enthalten sein sollen, bzw. keinen weiteren Block beanspruchen sollen [Kub18]. Arseny Kapoulkine hat verschiedene Meshletgrößen miteinander verglichen. Er ist zu dem Schluss gekommen, dass 64 Vertices und 84 Primitives am effizientesten ist, insbesondere dann, wenn im Task Shader Culling an den einzelnen Meshlets betrieben wird. Des weiteren ist die Empfehlung des Blogposts nach eigenen Tests zwar ein guter Maßstab, jedoch wird im Durchschnitt viel Speicher des Primitiven Buffers ungenutzt bleiben, da die 126 Primitiven mit 64 Vertices nie erreicht werden [Kap23].

4.3 Implementierung eines Standard Mesh Shaders

Wie im vorherigen Unterkapitel angekündigt, muss das Dreiecksnetz auf der CPU zu Meshlets geschnitten werden. Dazu wurde in dieser Arbeit der Meshoptimizer von Zeux verwendet [Zeu]. [Implementierung von Zeux beschreiben] Die Funktion „meshopt_buildMeshlets“ nimmt als Eingabeparameter die maximale Anzahl an Vertices und Primitiven (Kap.4.2), die Vertex und Index Daten sowie drei leere Buffer. Der Buffer *meshlet_indices* wird die neuen Index Daten enthalten, mit denen die Primitiven berechnet werden können. Der *meshlet_vertices* Buffer beinhaltet die einzigartigen Vertices des Dreiecksnetzes (Kap ??). Der letzte Buffer wird in dieser Arbeit als *Meshlet Descriptor* bezeichnet. Der Einfachheit halber wird er im Code jedoch einfach als *meshlets* implementiert. Der Meshlet Buffer setzt sich auch folgenden Elementen zusammen

- Vertex Count: Die Anzahl der Vertices V in dem Meshlet mit dem Index i
- Primitive Count: Die Anzahl der Primitives P in dem Meshlet mit dem Index i
- Vertex Offset: Die Menge an Schritten im Vertex Buffer, um an die Vertices des i -ten Meshlets zu gelangen
- Primitive Offset: Die Menge an Schritten im Index Buffer um an die Primitives des i -ten Meshlets zu gelangen

[BFB23] Im nächsten Abschnitt wird genauer auf die neuen Buffer eingegangen.

Vertex Index

Um auf einen Vertex zuzugreifen, wird der Buffer *meshlet_vertices* benötigt. Er beinhaltet Ganzzahlen ohne Vorzeichen, die auf einen bestimmten Vertex eines Meshlets zeigen. Im

Codeabschnitt 1 wird die Variable `vertexIndex` mittels der `GetVertexIndex` Methode gesetzt. In der Methode wird der lokale Vertex Index mittels:

$$localVertex = VertexOffset + localIndex$$

bestimmt. Mithilfe des `localIndex` kann nun der Vertex Index aus dem `meshlet_vertices` Buffer gelesen werden. Abschließend wird der Vertex mithilfe des Vertex Index aus dem Buffer mit den Vertex Ressourcen ausgelesen. Was hierbei festgestellt werden kann ist, dass eine doppelte Indexierung notwendig ist, und somit zwei Buffer ausgelesen werden müssen, um einen Vertex zu lesen. In einem späteren Abschnitt wird dieses Problem mithilfe von Duplizierung der Vertices gelöst.

Primitive Index

Der Buffer `meshlet_indices` ist für die Primitiven der Meshlets zuständig. Ein herkömmlicher Index Buffer enthält Ganzzahlen ohne Vorzeichen zwischen $0 - VertexCount - 1$. Die hier jedoch eine Indexierung auf Meshlet Ebene vorliegt, müssen diese Werte bei gleicher Buffergröße angepasst werden. Die Werte reichen nun anstatt von $0 - VertexCount$, von $0 - MaxPrimitiveCount - 1$. Das bedeutet, wenn Meshlets mit einer Vertex Anzahl von 64 und Primitiven Anzahl von 128 generiert werden, befinden sich in `meshlet_indices` lediglich Werte zwischen $0 - 127$. Um den aktuellen Index eines Meshlets zu erhalten muss ähnlich wie bei den Vertices der lokale Index berechnet werden. Dazu wird die Formel

$$localIndex = PrimitiveOffset + localIndex * indicesPerTriangle$$

verwendet. Da die Ausgabe einen 3-Dimensionalen Vektor für die Primitiven erwartet ist die Variable `indicesPerTriangle = 3`. Nun müssen nur noch die drei Indizes des aktuellen Meshlet Index aus dem Buffer gelesen und gesetzt werden.

Auffällig ist, dass die obere Schranke der Werte, die die Indizes enthalten, bedeutend geringer ist gegenüber eines herkömmlichen Index Buffers. Der benötigte Speicher eines einzelnen Indizes wird dadurch drastisch reduziert. Für einen Index wurden ursprünglich 4-Byte Speicher benötigt. Für die `meshlet_indices` werden in dem Fall von *V-Dach* = 128 und *I-Dach* = 256 nur 1 Byte pro Index verbraucht. Diese Auffälligkeit kann sich zunutze gemacht werden, indem drei Indizes, bzw. eine Primitive in ein 4-Byte Integer verpackt werden. Dadurch kann $\frac{2}{3}/66\%$ des Speicherbedarfs für den Index Buffer gespart werden.

Mit den Informationen der originalen Vertexdaten und der drei neu generierten Buffer `meshlet_indices`, `meshlet_vertices` und `meshlets`, kann nun der Mesh Shader gefüttert werden. Zunächst müssen die während des Build-Vorgangs kompilierten Shader gelesen werden. Diese

enthalten Informationen zum Layout der Root Signature, die daraufhin per API-Call erstellt wird. Bevor die Meshlet Daten an den Mesh Shader übergeben werden können, müssen diese in einen GPU Buffer geschrieben, und somit in den GPU RAM geschrieben werden. Wenn alles erledigt ist, können in der Commandlist der Constant Buffer und die benötigten Meshletdaten über die DirectX12 API-Calls `SetGraphicsRootConstantBuffer` und `SetGraphicsRootShaderResourceView` gesetzt werden.

4.4 Mesh Shader Implementation

Im Codeabschnitt 1 ist ein einfacher Mesh Shader zu sehen. Im Mesh Shader wird die Root Signature entsprechend den Anforderungen gesetzt. Minimal wird ein StructuredBuffer für jeden der auf der CPU generierten Meshlet Buffer benötigt. Um das Endresultat 3-Dimensional wirken zu lassen, wird ein ConstantBuffer verwendet, der die *model*, *modelView* und *modelViewProjection* Matrix beinhaltet. Zunächst wird das aktuelle Meshlet aus dem *Meshlet Descriptor Buffer* genommen. Die *SV_GroupID* stellt in dieser Implementierung den aktuellen Index der Meshlets dar. Um den lokalen Index des aktuellen Meshlets zu bekommen, muss die *SV_GroupThreadID* verwendet werden. Die aktuelle *GroupID* wird in einzelne Threads unterteilt, damit die GPU sich bei der parallelen Verarbeitung nicht in die queue kommt. Die Anzahl der Threads wird mittels $[NumThreads(128, 1, 1)]$ im Mesh Shader, oder, falls vorhanden, im Task Shader festgelegt.

Zu Beginn eines jeden Mesh Shaders muss die Anzahl der auszugebenden Vertices und Primitives mittels *SetMeshOutputCounts* gesetzt werden [Job19]. Dafür wurde eine Datenstruktur für den Meshlet Descriptor mit der Struktur wie sie in Kapitel. 4.3 definiert wurde, erstellt. Der Meshlet Descriptor mit der aktuellen *GroupID* wird also aus dem auf CPU Ebene erstellten Buffer gelesen und die Anzahl der Vertices und Primitives festgelegt. Welchen fundamentalen Nutzen der Meshlet Descriptor für einen Mesh Shader hat kann man hier erkennen. Die aktuellen *Group* wird in Threads aufgeteilt, die sich jeweils um einen Vertex und eine Primitive kümmern. Dazu wird überprüft ob sich die *GroupThreadID* noch innerhalb der Grenzen des aktuellen Meshlets befindet. Sollte dies der Fall sein, liest der Mesh Shader mithilfe des Meshlet Descriptors und der *GroupThreadID* Vertex und Primitive aus.


```
[RootSignature(ROOT_SIG)]
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void main(
    in uint localIndex : SV_GroupThreadID,
    in uint meshletIndex : SV_GroupID,
    out vertices VertexOut verts[64],
    out indices uint3 tris[128]
)
{
    Meshlet m = Meshlets[meshletIndex];

    SetMeshOutputCounts(m.VertCount, m.PrimCount);

    if (localIndex < m.PrimCount)
    {
        tris[localIndex] = GetPrimitive(m, localIndex);
    }

    if (localIndex < m.VertCount)
    {
        uint vertexIndex = GetVertexIndex(m, localIndex);
        verts[localIndex] = GetVertex(meshletIndex, vertexIndex);
    }
}
```

Code 1: Standard Mesh Shader main-Methode

4.5 Lokale Vertex Buffer

Der Mesh Shader im Codeabschnitt ?? zeigt eine Implementierung in seiner einfachsten Form. Der Plan ist jedoch, jedes Meshlet einzeln zu dekodieren, um die dekodierten Meshletdaten zu rendern. Der originale Vertex und Index Buffer müssen dafür angepasst werden. In Kap. 4.3 wurden bereits die benötigten Buffer erklärt, die der Meshoptimizer generiert. Vertex und Index Buffer werden mit diesem anpasst, damit jedes Meshlet über die gewünschte Geometrie und Topologie verfügt.

Um den Punkt der doppelten Indexierung des Vertex Buffers einzugehen, werden in diesem Abschnitt lokale Vertex Buffer für jedes Meshlet generiert. Die duplizierten Vertex Daten werden in Kauf genommen, damit sich während des Render Vorgangs ein zusätzlicher Elementzugriff

gespart werden kann.

Das Ziel ist es, die Vertex Daten auf die generierten Meshlets anzupassen, damit die Vertices der Meshlets sequentiell in einem Buffer liegen, wie es in Abb. 6 illustriert ist.

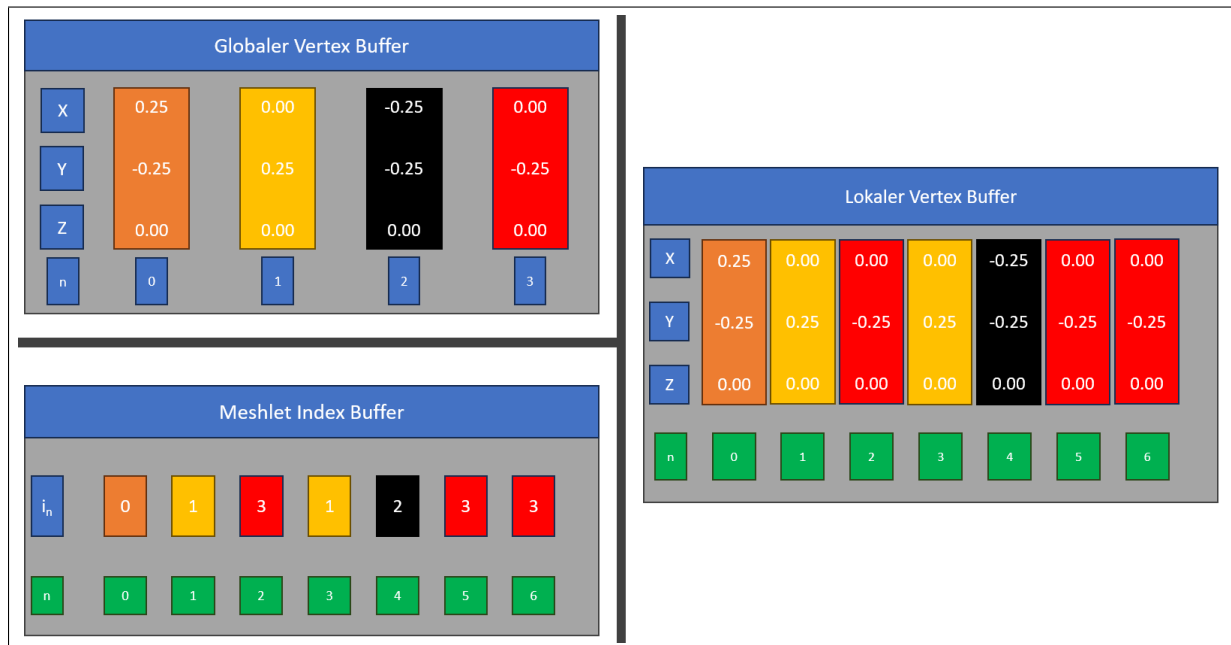


Abb. 6: **Lokaler Vertex Buffer** Die Abbildung zeigt eine Konstruktion eines Vertex Buffers der die Vertices aller Meshlets sequentiell beinhaltet.

In der Methode `GetVertexIndex` musste zunächst ein *Unique Vertex Index* berechnet werden, der den Index für den globalen Vertex Buffer bereitgestellt hat. Mithilfe des lokalen Vertex Buffers ist dieser Schritt nicht mehr nötig. Der korrekte Vertex Index kann nun lediglich mit dem Vertex Offset des Meshlets und der `GroupThreadID` berechnet werden. Mit diesem Index ist ein direkter Zugriff auf die Vertex Ressourcen möglich, ohne den *Unique Vertex Index* zu benötigen. Dadurch befinden sich zwar duplizierte Vertices im Vertex Buffer, dafür kann sich aber der ganze Buffer mit den *Unique Vertex Indices* gespart werden. Darauf folgt noch, dass der zusätzliche Elementzugriff des Wegfallenden Buffers nicht mehr notwendig ist, um zu den jeweiligen Meshlet Vertices zu gelangen. So besteht jeder einzelne, große Buffer mit allen Elementen sozusagen aus zusammengesetzten, kleinen Meshlet Buffern, die alle benötigten Elemente des jeweiligen Meshlets enthalten.

5 Kompressionsstandard Brotli-G

In vielen Anwendungsfällen wird auf Kombinationen von Kompressionsalgorithmen gesetzt. [QUELLE für Kompression] Auch Brotli macht sich mehrere Algorithmen zunutze. Google hat für die Entwicklung von Brotli eine eigene Implementierung des Deflate Algorithmus verwendet. Das Ergebnis der Kompression besteht aus einer Reihe an Meta Blöcken. Anstelle der kompletten Daten teilt Brotli den Datensatz in logische Blöcke ein, damit ein besseres Kompressionsergebnis erzielt werden kann. So wird jeder dieser Blöcke einzeln komprimiert, und zusammen mit den Header Informationen für die gesamten Daten in das Brotli Format geschrieben. Um die Daten zu komprimieren wird eine Kombination des LZ77 Algorithmus verwendet, um duplizierte Zeichenketten zu erkennen, und einer Huffman-Kodierung um Präfix freie Codewörter zu generieren. Jeder Meta Block hat dabei seine eigene Huffman-Kodierung. So sind Überschneidungen von Codewörtern in unterschiedlichen Meta Blöcken möglich, während ein Präfix freier Code in einem Meta Block gewährleistet ist. Der LZ77 Algorithmus hat zusätzlich noch die Möglichkeit, in einem zuvor kodierten Meta Block nach duplizierten Zeichenketten suchen, sollte diese Zeichenkette noch im Schiebefenster vorhanden sein [AS16].

Um eine parallele Dekompression zu ermöglichen, musste AMD bei Brotli-G einige Veränderungen an den Algorithmen vornehmen. Zum einen kann die Größe des Schiebefensters abweichen. Das kommt auf die Größe der Eingabedaten an. Um die Parallelisierung der Dekompression zu gewährleisten muss auf die Verwendung von vorherigen Meta Blöcken verzichtet, und das Schiebefenster zu Beginn eines neuen Meta Blocks zurücksetzen [AMD24].

Um die verwendeten Algorithmen besser zu verstehen werden sie in den folgenden Kapiteln genauer betrachtet.

5.1 LZ77

Der LZ77 (*Lempel-Ziv77*) Algorithmus gehört zu der Gruppe der Phrasenkodierung und ist ein verlustfreier, auf einem Wörterbuch basierender Algorithmus. Der Algorithmus komprimiert sequentielle Zeichenketten. Dabei kann dieser auf jeder Art von Daten, egal wie der Inhalt und die Größe aussieht, angewendet werden. Ob es sich lohnt, diesen anzuwenden, ist jedoch von den Daten abhängig. Beispielsweise sind Bilder ein schlechter Anwendungsfall, da sich die Informationen nur im Ausnahmefall wiederholen, und es deutlich bessere Kompressionsalgorithmen zur Komprimierung dieser gibt. Das Ziel des LZ77 Algorithmus ist lediglich, redundante Informationen zusammenzufassen.

Bevor der Algorithmus beschrieben wird, werden die benötigten Elemente definiert:

1. Eingabestrom: Die zu kodierenden Daten
2. Symbol: Ein willkürlich gewähltes Element des Eingabestroms
3. Datenfenster: Alle Symbole vom Start des Eingabestroms bis zum aktuell betrachteten Symbol
4. Vorschauenfenster: Ein Buffer fester Größe der Symbole vom aktuell betrachteten Symbol bis zum Ende des Buffers enthält
5. Schiebefenster: Daten- und Vorschauenfenster
6. Codewort: Ein Codewort bestehend aus dem Offset, der Lauflänge und des zu kodierenden Symbols

Der Ablauf des Algorithmus besteht aus folgenden Schritten:

Zu Beginn des Algorithmus wird das Datenfenster auf den Start des Eingabestroms gesetzt. Dieses Fenster ist zunächst leer. Das Vorschauenfenster wird vom Start des Eingabestroms mit Symbolen gefüllt, bis dieses voll ist. Zunächst wird das erste Symbol kodiert. Dafür verwendet der LZ77 Algorithmus ein Tupel in der Form von (*Position*, *Lauflänge*) und abschließend das zu kodierende Symbol. Dem Wörterbuch noch nicht bekannt Symbole werden neue Symbole mit (0, 0)Symbol hinzugefügt. Nach jedem Schritt wird das Schiebefenster um die Lauflänge der kodierten Symbole im Eingabestrom verschoben [Mic23].

5.1.1 Kodierung eines Codewortes

Zur Veranschaulichung wird das Codewort „laufenraufen“ mit dem LZ77 Algorithmus kodiert und anschließend wieder dekodiert. Daten- und Vorschauenfenster haben in diesem Beispiel eine Kapazität von jeweils sechs Symbolen.

Datenfenster	Vorschaufenster	restliches Codewort	Kodierung
	laufen	raufen	(0, 0)l
l	aufenr	aufen	(0, 0)a
la	ufenra	ufen	(0, 0)u
lau	fenrau	fen	(0, 0)f
lauf	enrauf	en	(0, 0)e
laufe	nraufe	n	(0, 0)n
laufen	raufen		(0, 0)r
aufenr	aufen		(6, 4)n

Tab. 1: LZ77 Kodierungs Beispiel

Eine Besonderheit die zunächst nicht intuitiv ist, ist die Konstruktion des letzten Codewortes in diesem Beispiel. Die Symbolsequenz „aufen“ mit der Kodierung $(6, 4)n$ könnte auch mit einem Offset von fünf kodiert werden. Im Normalfall würde die Symbolsequenz auch so kodiert werden. Da jedoch das Symbol „n“ das letzte Symbol des zu kodierenden Worts ist, und es so kein weiteres, zu kodierendes Symbol gibt, muss die Länge um eins reduziert werden, und das letzte Symbol kodiert werden.

Aus dem Beispiel geht hervor, dass die Auswahl der Buffergröße gut gewählt werden muss, damit der Algorithmus effektiv verwendet werden kann. Wäre in dem Beispiel das Datenfenster lediglich Platz für vier statt fünf Symbole, hätte die Symbolsequenz „aufe“ nicht als ganzes kodiert werden können. Die weiteren Iterationen der Lempel-Ziv Algorithmen haben statt einem lokalen Wörterbuch (Datenfenster) ein globales Wörterbuch verwendet. Durch die große Anzahl an Vergleichen erreicht der LZ77 Algorithmus ein besseres Kompressionsverhältnis als der LZ78 Algorithmus, benötigt für die Kompression jedoch länger. Wie lange das Komprimieren der Daten dauert ist jedoch nicht wichtig für diese Arbeit. Der interessante Punkt ist die Dekompressionsgeschwindigkeit. Der LZ77 Algorithmus ist bedeutend schneller bei der Dekomprimierung als bei der Komprimierung [CPP15].

5.1.2 Dekodierung eines Codeworts

In diesem Abschnitt soll aus der Kodierung das zuvor festgelegte Codewort dekodiert werden. Als Ergebnis aus der Kodierung erhalten wir den Ausgabestrom:

$(0, 0)l, (0, 0)a, (0, 0)u, (0, 0)f, (0, 0)e, (0, 0)n, (0, 0)r, (6, 4)n$.

Jetzt gilt es, das Datenfenster zu füllen. In einem Schritt der Dekodierung wird das Datenfenster überprüft, sollte Offset und Lauflänge ungleich 0 sein. Falls vorhanden werden die Daten des Datenfensters mit dem Symbol der aktuell zu dekodierenden Symbol dem Codewort hinzugefügt. Im nächsten Schritt wird die Symbolsequenz an die Symbole des Datenfensters verkettet, bis dieses voll ist. Anhand des letzten Schrittes der Dekompression sieht man wie der Algorithmus eine duplizierte Zeichenkette erkennt. In diesem Beispiel wird die Zeichenkette *aufe* mit einem Offset von 6 und einer Lauflänge von 4 aus dem Datenfenster gelesen, und mit dem Symbol *n* verkettet. Die Dekodierung ist nach diesem Schritt abgeschlossen.

Kodierung	Datenfenster	Codewort
$(0, 0)l$		l
$(0, 0)a$	l	la
$(0, 0)u$	la	lau
$(0, 0)f$	lau	lauf
$(0, 0)e$	lauf	laufe
$(0, 0)n$	laufe	laufen
$(0, 0)r$	laufen	laufenr
$(6, 4)n$	aufenr	laufenraufen
$\backslash 0$	raufen	laufenraufen

Tab. 2: Dekodierung mit dem LZ77-Algorithmus

Wie zu sehen ist, ist der LZ77 Algorithmus sowohl leicht verständlich als auch effektiv, wodurch er seinen Nutzen in vielen Anwendungen findet. Seine Weiterentwicklungen nennen sich LZ78 und LZW, die zwar dem LZ77 Algorithmus technisch überlegen sind, aufgrund von Patenten jedoch nicht so eine große Rolle spielen wie die erste Iteration von Lempel und Ziv. In Kombination mit anderen Techniken wie der Huffman Codierung bildet der LZ77-Algorithmus jedoch die Grundlage vieler leistungsstarker Kompressionsstandards, die heute in vielen Applikationen zu finden sind [QUELLE!!!].

5.2 Huffman-Kodierung

Eine gewisse Ähnlichkeit zu dem in der Einleitung angerissenen Thema des Morse Codes enthält die von Brotli verwendete Huffman-Kodierung. Die Huffman-Kodierung ist eine Methode zur verlustfreien Datenkompression und gehört zur Art der Codewort basierten Entropiekodierung. Ähnlich wie beim Morse Code, werden Symbole durch Bitfolgen substituiert. Was beim Morse Code als langes und kurzes Signal galt, ist im Huffman Code eine 0 oder 1. Mit der Huffman-Kodierung werden häufig auftretende Symbole durch kurze Bitfolgen dargestellt. Dementsprechend erhalten Symbole mit geringer Auftrittswahrscheinlichkeit ein langes Codewort. Bei der Betrachtung des Morse Codes fällt auf, dass nicht jeder Buchstabe die selbe Anzahl an Signalen beansprucht. Die Codewörter im Morse Code haben sich nämlich ebenfalls die Eigenschaft der Auftrittswahrscheinlichkeiten zunutze gemacht. Die im englischen Alphabet meist verwendeten Codewörter „E“ und „T“ werden beide mit jeweils einem Signal dargestellt. Das „E“ wird mit einem kurzen, während das „T“ vom langen Signal dargestellt wird. Mithilfe dieser Eigenschaft verbrauchen Symbole, die häufig auftreten, weniger Platz im Bitstrom [Mof19]. Histogramm

5.2.1 Konstruktion einer Huffman-Kodierung

Zur Konstruktion eines Huffman Codes wird ein Binärbaum generiert. Die zu kodierenden Symbole werden als Blätter des Baumes betrachtet. Der Baum wird sozusagen von „unten nach oben“ bzw. von „Blätter nach Wurzel“ aufgebaut.

In jedem Schritt werden die zwei Symbole oder Knoten mit der geringsten Auftrittswahrscheinlichkeit zu einem neuen Knoten verbunden. Die Auftrittswahrscheinlichkeit des neu erstellten Knotens ist die Summe der Auftrittswahrscheinlichkeiten der verbundenen Symbole/Knoten. Sobald die Wurzel des Baumes erreicht ist, also die Auftrittswahrscheinlichkeit bei 1 liegt, ist die Huffman-Kodierung abgeschlossen. Jedem Zweig des Baums wird zusätzlich eine 0 oder 1 zugewiesen. Ob der linke Kindknoten die 0 und der rechte Kindknoten die 1 bekommt, oder anders herum ist nicht wichtig, und kann von Implementierung zu Implementierung abweichen. Wichtig ist nur zu beachten, dass dies im gesamten Baum konsistent durchgezogen wird. Die Codewörter für jedes Symbol sind abzulesen, indem die Beschriftungen der Zweige als ein Bitstrom interpretiert werden, beginnend von der Wurzel.

Um den Vorteil dieser Eigenschaft zu Veranschaulichen, kann ein Vergleich mit dem *fixed length Code (FLC)* hilfreich sein. Anders als *Variable Length Code (VLC)* Verfahren wie die Huffman-Kodierung, wird jedem Symbol eines FLCs ein Codewort fester Länge zugewiesen. Die Auftrittswahrscheinlichkeit spielt bei der Erstellung von Codewörtern also keine Rolle. Um einen Vergleich zu ziehen kann die mittlere Codewortlänge des Alphabets, mit folgenden Symbolen betrachtet werden.

S_i	A	B	C	D
P_i	0.6	0.2	0.1	0.1

Die Formel zur Berechnung der mittleren Codewortlänge des FLCs lautet

$$\bar{l} = \lceil \log_2(N) \rceil$$

Wird das aus 4 Symbolen bestehende Beispieralphabet mittels FLC kodiert, ist die Codewortlänge l_i eines jeden Symbols = 2, wodurch auch die mittlere Codewortlänge bei 2,0 *Bits/Symbol* liegt.

Die Konstruktion des Binärbaums, aus dem die Codewörter entnommen werden können ist in Abb. 7 zu sehen.

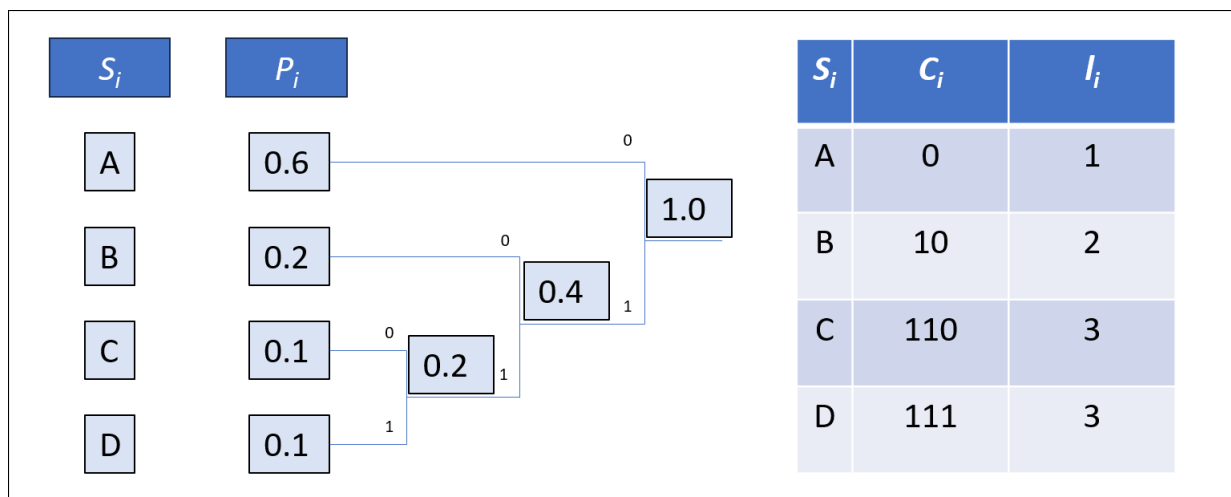


Abb. 7: **Erstellen eines Huffman Codes** Die Abbildung zeigt die Konstruktion eines Binärbaums und der daraus resultierenden Codewörter für die Symbole

5.2.2 Resultate einer Huffman-Kodierung

Aus diesem Beispiel kann der Nutzen der Huffman-Kodierung anhand von Werten ausgedrückt werden.

Um die mittlere Codewortlänge einer Huffman-Kodierung zu berechnen, wird die Formel

$$\bar{l} = \sum_{i=1}^n p_i \cdot l_i$$

benötigt

Aus dem Beispiel ergibt sich eine mittlere Codewortlänge von 1,6 *Bits/Symbol* bei einer Huffman-Kodierung. Im Vergleich zu einem FLC werden also 0,4 *Bits/Symbol* gespart. Die Formel für das Kompressionsverhältnis lautet:

$$C_r = \frac{\text{Eingabegröße}}{\text{Ausgabegröße}}$$

Da sich bei einer Huffman-Kodierung lediglich die Codewortlängen ändern, kann der FLC als Eingabe-, und der Huffman Code als Ausgabegröße in der Formel verwendet werden. So ergibt sich ein Kompressionsverhältnis von:

$$C_r = \frac{\lceil \log_2(N) \rceil}{\sum_{i=1}^n p_i \cdot l_i}$$

$$C_r = \frac{2,0 \text{ Bits/Symbol}}{1,6 \text{ Bits/Symbol}} = 1,25$$

Die Effektivität der Huffman-Kodierung hängt stark von der Verteilung der Symbole in der Datenquelle ab. Wenn der konstruierte Binärbaum stark balanciert ist, bedeutet dies, dass die Codewörter für die einzelnen Symbole ähnliche Längen haben. In solchen Fällen ist die Huffman-Kodierung weniger effektiv, da sie weniger Redundanz in den Daten ausnutzen kann.

Redundanz beschreibt den unnötigen Aufwand zu Repräsentation einer Information. Wenn man eine Quelle, wie beispielsweise ein Alphabet betrachtet, spricht man von der Redundanz einer Quelle oder *Quellredundanz*. Die Quellredundanz ist abhängig von dem Entscheidungsgehalt H_0 des Alphabets und der Quellentropie $H(X)$.

$$\Delta R_0 = H_0 - H(X)$$

$$H_0 = \sum_{i=1}^n p_i * \log_2 p_i, \quad p_i = \frac{1}{n} \quad \forall i$$

$$H_0 = \sum_{i=1}^n p_i * \log_2 p_i, \quad p_i = \frac{1}{n} \quad \forall i$$

Die Aufgabe der Huffman-Kodierung ist die Minimierung der *Codierungsredundanz*.

Im Gegensatz dazu profitiert die Huffman-Kodierung enorm von einer ungleichen Verteilung der Symbole, bei der bestimmte Symbole deutlich häufiger auftreten als andere. Dadurch können kürzere Codewörter für häufige Symbole und längere Codewörter für seltene Symbole verwendet werden, was zu einer besseren Kompression führt.

Allerdings ist die Huffman-Kodierung anfällig für Veränderungen in der statistischen Verteilung der Symbole. Wenn sich die Auftrittswahrscheinlichkeiten im Laufe der Anwendung ändern oder wenn sie falsch berechnet wurden, kann dies zu einer Verschlechterung der Kompression führen. Im Extremfall kann es vorkommen, dass das Symbol mit der höchsten Auftrittswahrscheinlichkeit

das längste Codewort erhält, was die Effizienz der Codierung erheblich beeinträchtigt. Daher ist es wichtig, dass die Statistik der Symbole regelmäßig überprüft und aktualisiert wird. Um das zu bewerkstelligen kann eine adaptive Huffman-Kodierung verwendet werden. Anstelle einer festen Symbolstatistik wird bei der adaptiven Huffman-Kodierung die Statistik nach dem Kodieren eines Symbols überprüft und aktualisiert. Adaptive Algorithmen liefern zu Beginn einer Kodierung noch keine guten Ergebnisse, da die Symbolstatistik bei wenigen Symbolen nicht Aussagekräftig ist [JPJ98].

5.3 GZIP

Brotli-G misst sich mit anderen Kompressionsstandards wie GZip und Deflate. Der Deflate Algorithmus beschreibt eine Kombination aus dem LZ77 Algorithmus und der Huffman-Kodierung [Wahrscheinlich kein eigenes Kapitel. Lieber Deflate Algorithmus in anderen Kapiteln erwähnen]

5.4 Brotli-G Compute Shader

Das komprimierte Dreiecksnetz soll mittels Brotli-G's GPU Dekodierer dekodiert werden. Da Brotli-G mit der Grafik-API DirectX12 arbeitet, wird eine GPU benötigt die das Shader Model 6 unterstützt. Für die Verwendung des GPU Dekodierers wird also eine NVidia Grafikkarte mit der Turing-Architektur benötigt [Bur20]. Diese erkennt man am RTX Präfix vor der Modellnummer. Bei AMD Grafikkarten muss mindestens die RDNA-2 Architektur verbaut sein. Mit Grafikkarten aus der AMD RX6000er Reihe für RDNA-2, und der RX7000er Reihe für RDNA-3 kann der GPU Dekodierer verwendet werden. Brotli-G definiert ihren GPU Dekodierer in der Shader-Sprache *high-level shader language (HLSL)* und ist als Compute Shader definiert (Kap. 2.4).

Da es sich bei Brotli-G um eine Modifikation des Brotli Formats handelt, wurde der Bitstrom des Kodierers angepasst.

5.5 CPU Ebene

Queries Buffers etc

6 Ergebnisse

Dieses Kapitel widmet sich den Ergebnissen dieser Arbeit. Im Kap. 3.1 wurde eine grobe Schilderung gegeben, wie die Pipeline aussieht, die ein beliebiges 3D-Modell durchläuft, bis es gezeichnet wird. Im ersten Anlauf wurde ein Datensatz bestehend aus 11 unterschiedlichen 3D-Modellen durch diese Pipeline geschickt. Dabei wurde der Kompressionsstandard Brotli-G untersucht. Wichtige Gesichtspunkte für die Auswertung sind das Kompressionsverhältnis, und die Geschwindigkeit, in der die komprimierten Dreiecksnetze dekomprimiert werden. Die visuelle Qualität muss im ersten Anlauf zwangsweise unverändert bleiben, da Brotli-G verlustfrei komprimiert (Kap. 2.1). Die Ergebnisse des ersten Ablaufs sind in der nachfolgenden Abb.(einfügen) dokumentiert.


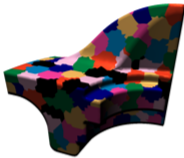


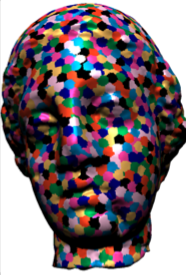

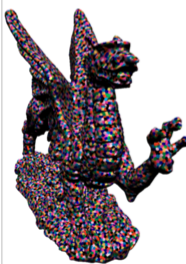




Rockerarm	Fandisk	Hand	Horse	Igea	Isis
					
Vertices 53.967 Indices 242.348 Meshlets 844	Vertices 8.675 Indices 39.040 Meshlets 136	Vertices 437.283 Indices 1.974.540 Meshlets 6.833	Vertices 65.029 Indices 292.472 Meshlets 1.017	Vertices 180.672 Indices 810.096 Meshlets 2.824	Vertices 251.363 Indices 1.128.932 Meshlets 3.928
Welsh Dragon	Angel	Armadillo	Bunny	Dinosaur	
					
Vertices 1.473.246 Indices 6.669.964 Meshlets 23.021	Vertices 317.440 Indices 1.429.640 Meshlets 4.961	Vertices 231.081 Indices 1.043.412 Meshlets 3.611	Vertices 46.930 Indices 210.120 Meshlets 734	Vertices 75.189 Indices 338.944 Meshlets 1.175	

Abb. 8: **Der verwendete Datensatz** Die Abbildung zeigt alle Dreiecksnetze mit der Anzahl an Vertices/Indizes/Meshlets.

Der Datensatz besteht aus vielen unterschiedlichen 3D-Modellen unterschiedlicher Größe. Von einem sehr kleinem Modell der „Fandisk“ mit 136 Meshlets, bis hin zu einem „Welsh Dragon“, der aus ganzen 23.021 Meshlets besteht, werden nun die Ergebnisse des Brotli-G Kodierers ausgewertet und analysiert.

6.1 Auswertung des Datensatzes

Betrachten wir zunächst das Dreiecksnetz mit der geringsten Anzahl an Vertices/Indizes/Meshlets. Die Ergebnisse des Kodierers sind sehr vielversprechend. Die 366.536 Bytes des Originalmodells komprimiert Brotli-G auf 160.200 Bytes, und erreicht somit ein Kompressionsverhältnis von 2,29. Auffällig sind hierbei jedoch die Dekompressionszeiten für dieses sehr kleine Modell. Bei der Größe des Modells und der doch sehr hohen Dekompressionszeit für den GPU Dekodierer wird eine Bandbreite von 0,054 GiB/s erreicht, was deutlich weniger ist, als was der PCIE-Bus der GPU zulässt, die für den Versuch verwendet wurde.

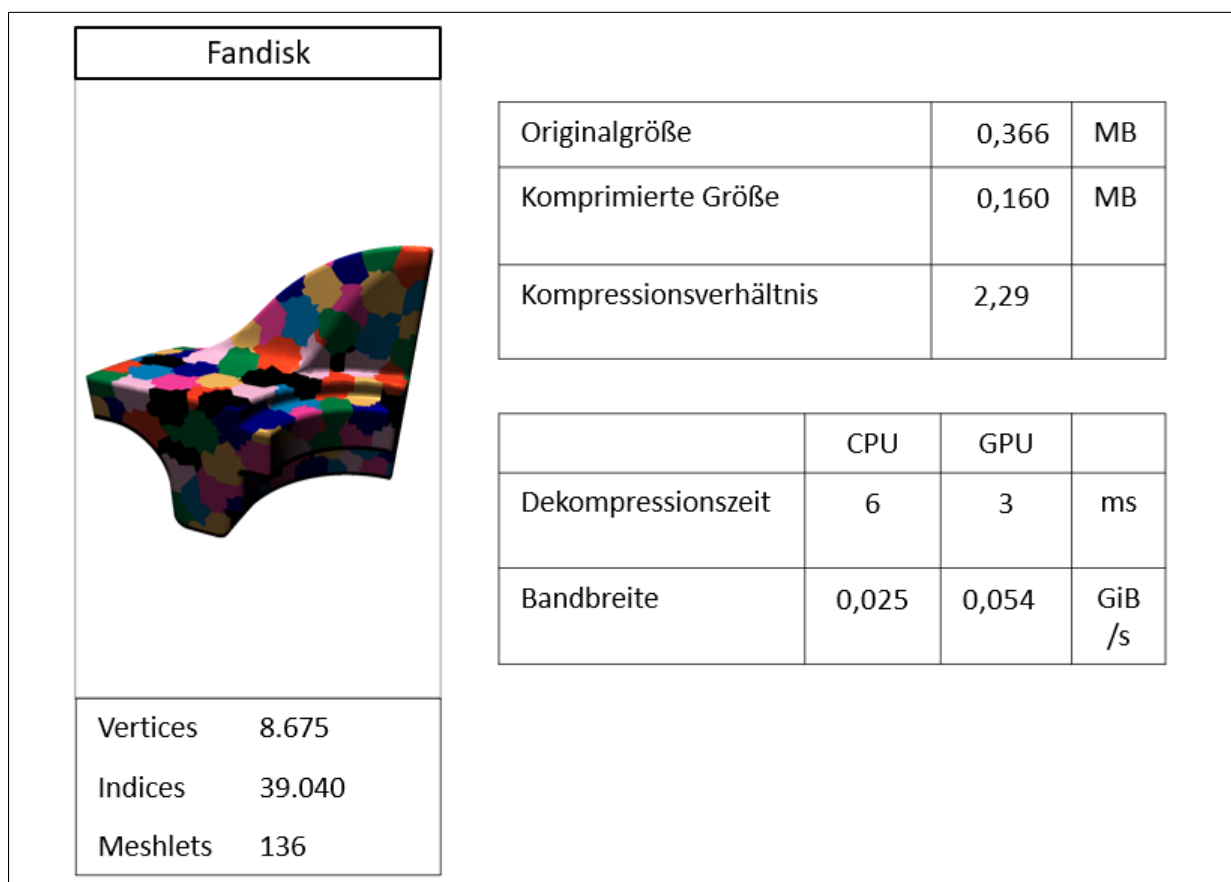


Abb. 9: **Fandisk Kompressionsergebnis** Das kleinste Dreiecksnetz aus dem Datensatz, und seine Ergebnisse

Um die niedrige Bandbreite zu erklären, können die Ergebnisse des Welsh Dragons betrachtet werden. Dieser besteht aus sehr viel mehr Vertices/Indizes/Meshlets, was sich in der Größe des Modells widerspiegelt. Der Welsh Dragon hat eine Originalgröße von 62406096 Bytes, die auf 31716764 Bytes komprimiert wurde. Das entspricht einem Kompressionsverhältnis von 1,97. Viel interessanter hierbei sind jedoch die Dekompressionszeit und die daraus resultierende Bandbreite. Der GPU Brotli-G GPU Dekodierer benötigt nicht unbedingt viel mehr Zeit für den Welsh Dragon (7 ms) gegenüber der Fandisk (3 ms). Bei Betrachtung der komprimierten Größe der beiden Dreiecksnetze fällt auf, dass der Welsh Dragon sehr viel mehr Daten benötigt. Während die

Dekompression des Welsh Dragons etwas mehr als das zweifache der Zeit der Fandisk benötigt, ist die komprimierte Größe des Welsh Dragons etwa 200x so groß wie diese. Das Ergebnis davon ist in der Bandbreite zu sehen, die beim Welsh Dragon sehr viel besser ist, jedoch noch immer sehr weit von den gewünschten Werten entfernt ist. Die parallele Dekodierung des Brotli-G Dekodierers scheint daher erst bei größeren Datensätzen richtig effektiv zu werden.

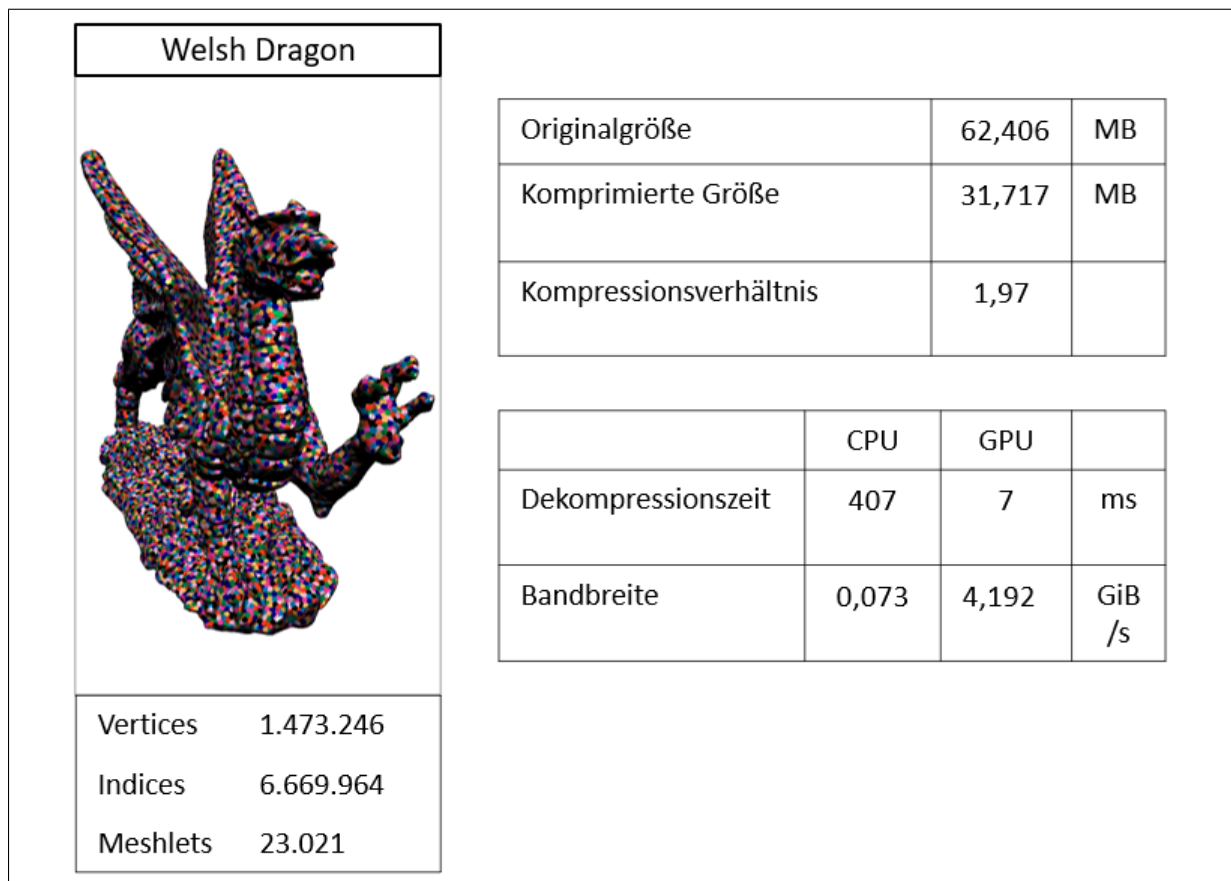


Abb. 10: **Welsh Dragon Kompressionsergebnis** Das größte Dreiecksnetz aus dem Datensatz, und seine Ergebnisse

6.2 Auswertung der quantisierten Vertex-Daten

Im zweiten Anlauf werden die Vertex Daten vor der Komprimierung zu 16-Bit floating point values quantisiert. Ziel davon ist, die niederwertigen Bits, die nur noch wenig zur Struktur des 3D-Modells beitragen, loszuwerden. Die hinteren 16 Bit einer 32 Bit Gleitkommazahl beansprucht genausoviel Speicher wie die vorderen 16 Bit. Während die vorderen Bits jedoch die grobe Position des Vertex, oder auch die Richtung der Normalen, sind die niederwertigen Bits für sehr feine details wichtig. Wenn es jedoch nicht gerade in Richtung von medizinischen Anwendungen geht, oder allgemeiner beschrieben in Bereiche, in denen diese feine Granularität sehr wichtig ist, sind 16 Bit pro Komponente ausreichend, damit ein Dreiecks visuell ansprechend bleibt.

Um die Qualität zu visualisieren, sieht man in Abb. 11 das Stanford Bunny einmal mit 16, und einmal mit 32 Bit Vertex Attributen.

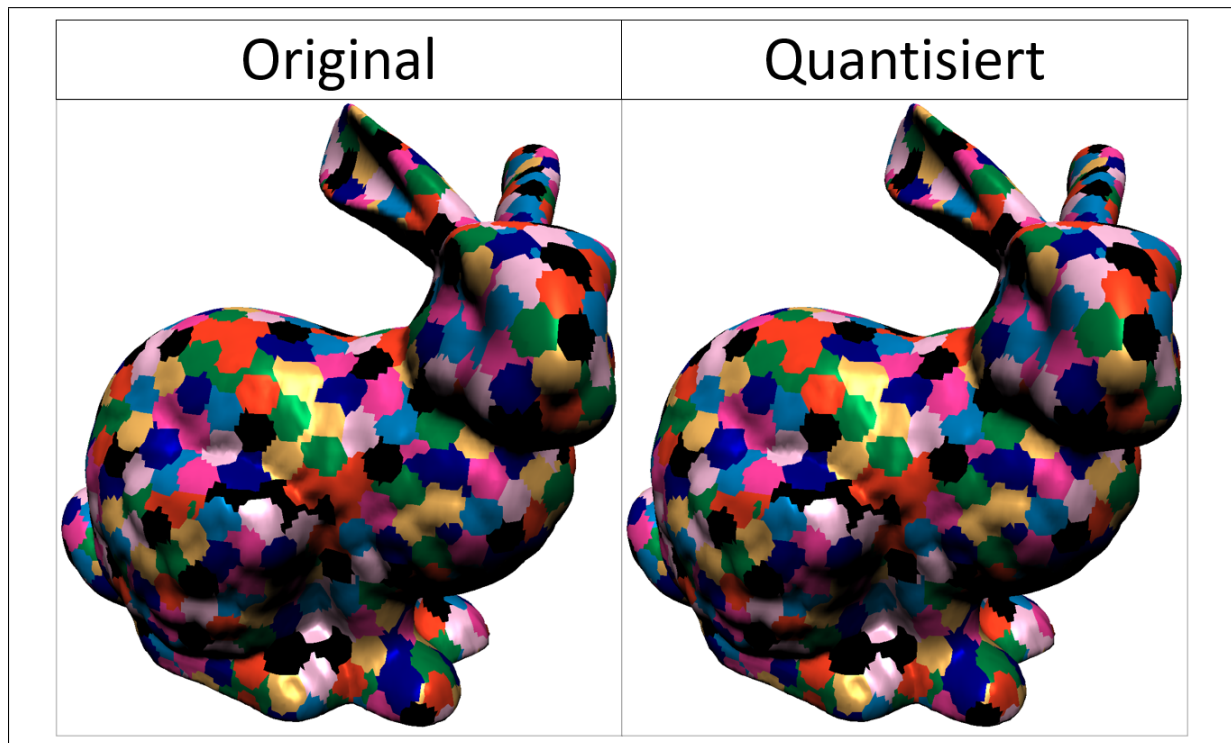


Abb. 11: **Quantisiertes Stanford Bunny** Die Abbildung zeigt eine Gegenüberstellung des Bunnys mit 32 Bit Vertex Attributen und dem Bunny mit den quantisierten 16 Bit Vertex Attributen

Bei der Gegenüberstellung sind die Unterschiede kaum erkennbar, was sehr hilfreich ist, wenn die Kompression mit Brotli-G betrachtet wird. Um einen Vergleich der Ergebnisse zu ziehen wird wieder der Welsh Dragon zur Auswertung der Kompressionsergebnisse verwendet. Lediglich die Quantisierung der Vertex Daten spart rund 29% der Größe vor der Komprimierung. Die Originalgröße von 44.727.144 Bytes wird anschließend von Brotli-G auf 17.260.728 Bytes komprimiert, was einem Kompressionsverhältnis von 2,59 entspricht. Die Dekomprimierung auf der GPU nimmt mit 6 ms in etwa soviel Zeit in Anspruch wie der Welsh Dragon mit 32 Bit Gleitkommazahlen. Dadurch ergibt sich eine Bandbreite von 2,706 GiB/s.

Im Anhang sind die Ergebnisse für alle Dreiecksnetze aus dem Datensatz ersichtlich. Dabei fällt auf, dass der Welsh Dragon bei der Betrachtung der Kompressionsverhältnisse kein Einzelfall ist. Betrachtet man die Ergebnisse von quantisierten und nicht quantisierten Dreiecksnetzen, ist klar zu erkennen, dass die quantisierten Dreiecksnetze nicht nur weniger Speicherplatz beanspruchen. Das Kompressionsverhältnis ist im Durchschnitt des Datensatzes bei den quantisierten Dreiecksnetzen um 18,82%.

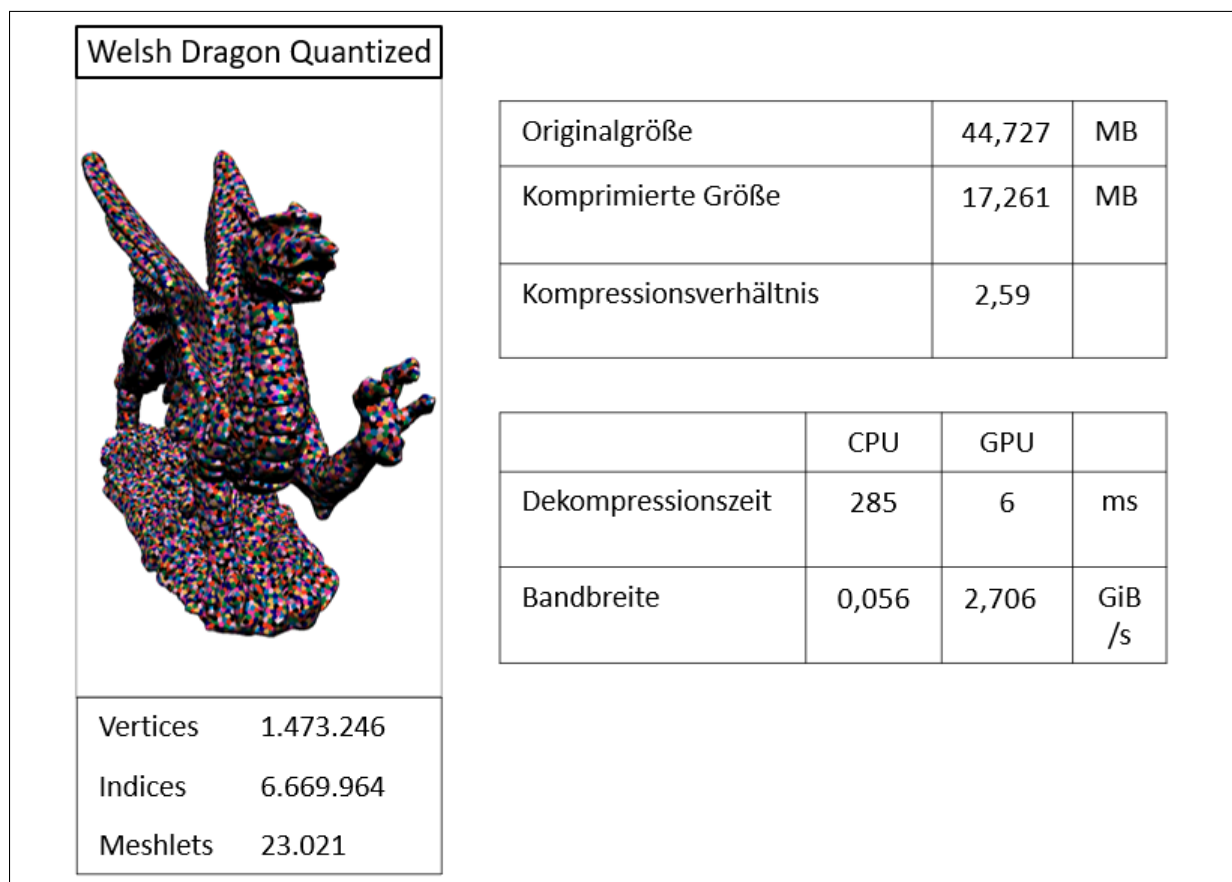



Abb. 12: **Welsh Dragon Kompressionsergebnis quantisiert** Die Ergebnisse des quantisierten Welsh Dragon

6.3 Auswertung eines großen Dreiecksnetzes

Wie in den anderen Versuchen erkannt wurde, wird eine bestmögliche Bandbreite bei großen Dreiecksnetzen erreicht. Dafür wird in diesem Abschnitt ein Teil eines großen Dreiecksnetzes des Davids ausgewertet. Leider legt Brotli-G eine maximale GPU Buffer Größe von 1 GB fest, wodurch ein maximal zu dekodierender Block eingeschränkt wird. Sollte ein größeres Dreiecksnetz dekodiert werden, muss dieses sinngemäß zugeschnitten werden. Die Haare des David bestehen aus fünf kleinen Teilstücken, die zusammengetragen eine Größe von rund 760 MB aufweisen, und so einen Großteil der maximalen Buffergröße verwenden. Die kleinen Teilstücke wurden auf der CPU nacheinander eingelesen, und in ein gemeinsames Binärobjekt geschrieben, das von Brotli-G kodiert worden ist.

Brotli-G hat die Größe des Davids von 759.567.128 Bytes auf 326.929.888 Bytes komprimiert, und somit ein Kompressionsverhältnis von 2,32 erreicht. Auf der GPU wurde dieses in 65 ms dekomprimiert, wodurch eine Bandbreite von 4,682 GiB/s erreicht wurde.

Auch bei diesem Dreiecksnetz wird zusätzlich noch ein Ergebnis mit quantisierten Vertex Attributen durchgeführt. Durch die Quantisierung der Vertex Attribute kann beim David etwas mehr als 200 MB an Speicher reduzieren.

David	
	
Vertices	17.963.799
Indices	68.000.204
Meshlets	280.706

Originalgröße	759,57	MB
Komprimierte Größe	326,93	MB
Kompressionsverhältnis	2,32	

	CPU	GPU	
Dekompressionszeit	17.880	65	ms
Bandbreite	0,017	4,682	GiB /s

Abb. 13: **David Kompressionsergebnis** Ein Teil von dem großen Dreiecksnetzes des Davids

Das Kompressionsverhältnis des quantisierten Davids ist mit 4,67 sehr gut, und dadurch zurückzuführen, das der Scan sehr detailliert

David Quantized

Vertices	17.963.799
Indices	68.000.204
Meshlets	280.706

Originalgröße

544,00

MB

Komprimierte Größe

115,29

MB

Kompressionsverhältnis

4,72

Dekompressionszeit

5.842

37

ms

Bandbreite

0,018

2,893

GiB
/s

Abb. 14: **David Kompressionsergebnis quantisiert** Ein Teil von dem großen Dreiecksnetzes des Davids mit quantisierten Vertex Attributen

7 Fazit

Für die Versuche wurde eine *RX 6650 XT* von AMD verwendet. Diese kann unter optimalen Bedingungen eine maximale Bandbreite von 238 GiB/s erreichen. Die Ergebnisse sind

Wie in den Ergebnissen zu sehen bietet die Quantisierung dazu noch eine nicht zu verachtende Reduktion der Speichergröße. Die Kombination mit der Kompression mit Brotli-G bietet dazu noch bessere Ergebnisse der komprimierten Größe, aufgrund von dem nicht kodieren müssen der niederwertigen Bits, die nur rauschen verursachen.

7.1 Ausblick

Literaturverzeichnis

- [AMD22] AMD: Brotli-G: An open-source compression/decompression standard for digital assets that is compatible with GPU hardware. In: *AMD GPUOpen* (2022). <https://gpuopen.com/brotli-g-sdk-announce/>
- [AMD24] AMD: Brotli-G Bitstream Format. (2024)
- [AS16] ALAKUIJALA, Jyrki ; SZABADKA, Zoltan: *Brotli Compressed Data Format*. RFC 7932. <http://dx.doi.org/10.17487/RFC7932>. Version: Juli 2016 (Request for Comments)
- [BFB23] BO, Jensen M. ; FRISVAD, Jeppe R. ; BÆRENTZEN, J. A.: Performance Comparison of Meshlet Generation Strategies. In: *Journal of Computer Graphics Techniques* (2023)
- [Bur20] BURGESS, John: RTX on—The NVIDIA Turing GPU. In: *IEEE Micro* 40 (2020), Nr. 2, S. 36–44. <http://dx.doi.org/10.1109/MM.2020.2971677>. – DOI 10.1109/MM.2020.2971677
- [Car22] CARVALHO, Miguel Ângelo Abreu d.: Exploring Mesh Shaders. (2022)
- [CPP15] CHOUDHARY, Suman M. ; PATEL, Anjali S. ; PARMAR, Sonal J.: Study of LZ 77 and LZ 78 Data Compression Techniques, 2015
- [CR12] COZZI, P. ; RICCIO, C.: *OpenGL Insights*. Taylor & Francis, 2012 (Online access with subscription: Proquest Ebook Central). <https://books.google.de/books?id=CCVenzOGjpcC>. – ISBN 9781439893760
- [DC96] In: DAL CIN, Mario: *Klassifizierung von Rechnerarchitekturen*. Wiesbaden : Vieweg+Teubner Verlag, 1996. – ISBN 978-3-322-94769-7, 22–32
- [Ile22] In: ILETT, Daniel: *Advanced Shaders*. Berkeley, CA : Apress, 2022. – ISBN 978-1-4842-8652-4, 517–582
- [JBG17] JAKOB, Johannes ; BUCHENAU, Christoph ; GUTHE, Michael: A Parallel Approach to Compression and Decompression of Triangle Meshes using the GPU. In: *Computer Graphics Forum* 36 (2017), Nr. 5, 71-80. <http://dx.doi.org/https://doi.org/10.1111/cgf.13246>. – DOI <https://doi.org/10.1111/cgf.13246>
- [Job19] JOBALIA, Sarah: Coming to DirectX 12— Mesh Shaders and Amplification Shaders: Reinventing the Geometry Pipeline. In: *DirectX Developer Blog* (2019). <https://devblogs.microsoft.com/directx/coming-to-directx-12-mesh-shaders-and-amplification-shaders-reinventing-the-geometry-pipeline/>

- [JPJ98] JEON, Byeungwoo ; PARK, Juha ; JEONG, Jechang: Huffman coding of DCT coefficients using dynamic codeword assignment and adaptive codebook selection. In: *Signal Processing: Image Communication* 12 (1998), Nr. 3, 253-262. [http://dx.doi.org/https://doi.org/10.1016/S0923-5965\(97\)00041-6](http://dx.doi.org/https://doi.org/10.1016/S0923-5965(97)00041-6). – DOI [https://doi.org/10.1016/S0923-5965\(97\)00041-6](https://doi.org/10.1016/S0923-5965(97)00041-6). – ISSN 0923-5965
- [Kap23] KAPOULKINE, Arseny: Meshlet size tradeoffs. (2023). <https://zeux.io/2023/01/16/meshlet-size-tradeoffs/>
- [Kub18] KUBISCH, Christoph: Introduction to Turing Mesh Shaders. (2018). <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/#entry-content-comments>
- [Mic21] MICROSOFT: Typed Unordered Access Views (UAV). In: *Microsoft Documentation* (2021)
- [Mic23] MICROSOFT: LZ77 Compression Algorithm. In: *Microsoft Documentation* (2023). https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6ceff1cccb
- [Mof19] MOFFAT, Alistair: Huffman Coding. In: *ACM Comput. Surv.* 52 (2019), aug, Nr. 4. <http://dx.doi.org/10.1145/3342555>. – DOI 10.1145/3342555. – ISSN 0360-0300
- [Ope24] OpenGL: Compute Shader. In: *OpenGL Wiki* ((besucht am 09.02.2024)). https://www.khronos.org/opengl/wiki/Compute_Shader
- [Zeu] ZEUX: meshoptimizer. <https://github.com/zeux/meshoptimizer>

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Titel

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

Ort

Datum

Unterschrift