



Hochschule für angewandte Wissenschaften Coburg
Fakultät Elektrotechnik und Informatik

Studiengang: Informatik

Bachelorarbeit

Geometriedekompression von Dreiecksnetzen auf der GPU

Janek Foote

Abgabe der Arbeit: 12.04.2024

Betreut durch:

Quirin Meyer, Hochschule Coburg

Inhaltsverzeichnis

Abbildungsverzeichnis	4
Tabellenverzeichnis	5
Codebeispielverzeichnis	6
1 Einführung	8
1.1 Geschichtlicher Hintergrund der Datenkompression	8
1.2 Steigende Komplexität	10
1.3 Ziel der Arbeit	11
2 Grundlagen	13
2.1 Brotli Kompressionsstandard	13
2.2 Parallelle Datenverarbeitung	13
2.3 Die traditionelle Rendering Pipeline	16
2.3.1 Vertex Shader	16
2.3.2 Tessellation Stage	17
2.3.3 Geometry Shader	17
2.3.4 Pixel Shader	18
2.4 Compute Shader	18
2.5 Quantisierung von Gleitkommazahlen	19
2.6 Grundbegriffe der Datenkompression	21
3 Methodik	24
4 Mesh Shader	26
4.1 Die Mesh Shading Pipeline	26
4.2 Meshlets	27
4.3 Implementierung eines Standard Mesh Shaders	28
4.4 Mesh Shader Implementation	30
4.5 Lokale Vertex-Buffer	32
5 Kompressionsstandard Brotli-G	34
5.1 LZ77	34
5.1.1 Kodierung eines Codewortes	35
5.1.2 Dekodierung eines Codeworts	37
5.2 Huffman-Kodierung	38
5.2.1 Konstruktion einer Huffman-Kodierung	38
5.2.2 Resultate einer Huffman-Kodierung	39

Inhaltsverzeichnis

6 Ergebnisse	42
6.1 Die Ergebnisse des Datensatzes	43
6.2 Auswertung der quantisierten Vertex-Daten	44
6.3 Die Resultate von Michelangelos David	46
7 Fazit	49
7.1 Analyse der Dekompressionszeit und Bandbreite	49
7.2 Analyse der Kompressionsverhältnisse	50
7.3 Analyse der quantisierten Dreiecksnetze	50
7.4 Ausblick für spätere Arbeiten	52
Literaturverzeichnis	54
A1 Anhang	57
A1.1 Ergebnisse ohne Quantisierung	57
A1.2 Ergebnisse mit Quantisierung	64
A1.3 Berechnung der verwendeten Prozentzahlen	71
Ehrenwörtliche Erklärung	

Abbildungsverzeichnis

Abb. 1:	YouTube Statistik	9
Abb. 2:	SIMD Pattern	14
Abb. 3:	traditionelle Rendering Pipeline	16
Abb. 4:	Repräsentation einer Zahl	20
Abb. 5:	Flussdiagramm des Ablaufs	25
Abb. 6:	Mesh Shading Pipeline	26
Abb. 7:	Lokaler Vertex-Buffer	32
Abb. 8:	Huffman Code Beispiel	39
Abb. 9:	Der verwendete Datensatz	42
Abb. 10:	Fandisk Kompressionsergebnis	43
Abb. 11:	Welsh Dragon Kompressionsergebnis	44
Abb. 12:	Quantisiertes Stanford Bunny	45
Abb. 13:	Welsh Dragon Kompressionsergebnis quantisiert	46
Abb. 14:	David Kompressionsergebnis	47
Abb. 15:	David Kompressionsergebnis quantisiert	48
Abb. 16:	Mid-Tread Quantisierer	51
Abb. 17:	NVIDIAs nvCOMP Testergebnisse	53

Tabellenverzeichnis

Tab. 1:	LZ77-Kodierung Beispiel	36
Tab. 2:	Dekodierung mit dem LZ77-Algorithmus	37

Codebeispielverzeichnis

Code 1:	Quantisierung von Float zu Half	21
Code 2:	Main Methode des Mesh Shaders	31

Zusammenfassung

In der Videospiel- und Animationsfilmindustrie spielen 3D-Modelle eine zentrale Rolle. Sie bieten einem Künstler die Möglichkeit Charakteren und Umgebungen zu modellieren, die eine Darstellung der Realität simulieren soll.

In der Computergrafik sind Dreiecksnetze eine gängige Datenrepräsentation von 3D-Modellen. Die Modelle werden jedoch immer komplexer, was den Speicherbedarf erhöht und die zur Darstellung benötigte Zeit verlängert. Um diesen Anforderungen gerecht zu werden, müssen neue Methoden zur Datenkompression entwickelt werden. Ziel dieser Arbeit ist es, ein Verfahren zu entwickeln, um ein komprimiertes Dreiecksnetz in die GPU zu laden und dort zu dekodieren. Insbesondere sollen das Kompressionsverhältnis, Dekompressionsrate und visuelle Qualität quantitativ untersucht und auswertet werden. Die Ergebnisse dieser Arbeit werden, wenn möglich, mit den Resultaten von gängigen Kompressionsmethoden verglichen.

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 Normalen und Texturkoordinaten/Farbwerte für jeden Punkt des Dreiecksnetzes in 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, realitätsgerecht 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 Geschichtlicher Hintergrund der 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 Wörter und 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 des Alphabets). 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 für andere Alphabete 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 Vorteile von komprimierten Daten in der modernen Welt. Die meisten Menschen denken an JPEG und PNG, 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 in der Industrie verwendeten Betriebssystems komprimiert diese Speichern von Daten automatisch. Zusätzlich dazu besteht noch die Möglichkeit, seine Daten manuell zu komprimieren mithilfe von Programmen wie 7-Zip, 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 (siehe Abbildung 1). Um die Unmengen an Videos zu speichern, benötigt Google riesige Serverfarmen, die auf

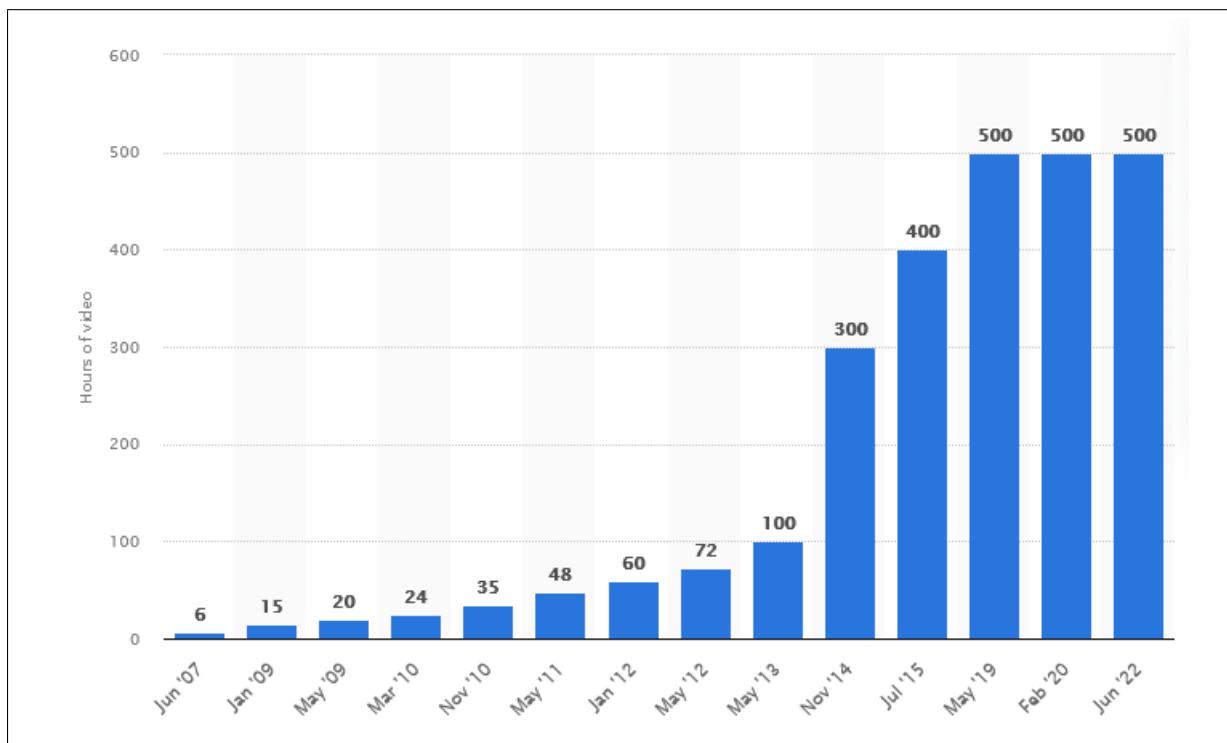


Abb. 1: **Anzahl der YouTube-Videos** Die Anzahl an Minuten, die auf YouTube hochgeladen werden. Abbildung von Statista:

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

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, würden die Daten nicht

komprimiert werden.

Ein weiterer Aspekt ist der eigentliche Nutzen von YouTube, nämlich das Abspielen von Videos. Um ein Video sehen zu können, muss es vom YouTube-Server 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 Trancoding, 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 Komprimieren und Dekomprimieren 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 einer Auflösung von 4K wiedergegeben werden soll. Trotzdem werden die vollen Daten des Videoformats empfangen. Um diese Verschwendungen 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 bei geringerer Bitrate beibehalten werden. 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

Das Beispiel der Unterhaltungsbranche hat aufgezeigt, warum Kompression in unserer heutigen Gesellschaft eine große Rolle spielt. Um die Relevanz von Kompressionsalgorithmen in dem Kontext von Dreiecksnetzen aufzuzeigen, kann sich ein Beispiel an Videospielen genommen werden. Physische Medien waren zu Beginn des 21. Jahrhunderts der Standard, um eine Form der Unterhaltung in den Haushalt zu bekommen. Sei es die Kassette um Musik zu hören, die

VHS um einen Film zu sehen oder die CD um ein neues Spiel zu spielen. All diese physischen Medien haben jedoch nur eine begrenzte Anzahl an Speicherplatz. Um das auf dem Cover versprochene Produkt zu liefern, haben die Hersteller zwei Optionen. Die Daten können auf mehrere Speichermedien aufgeteilt oder komprimiert werden. Die zwei Optionen schließen sich jedoch nicht gegenseitig aus. In den allermeisten Fällen mussten selbst die komprimierten Dateien auf unterschiedliche Speichermedien aufgeteilt werden. Neben all den anderen Assets nehmen Dreiecksnetze in 3D-Spielen einen beachtlichen Anteil an Speicherplatz ein. So gut wie alles sichtbare in einem 3D-Spiel besteht aus Dreiecksnetzen, beginnend bei Strukturen wie Gebäuden, Spieler und *non-player Characters (NPCs)*, oder das Ambiente mit Kisten, Blumentöpfen und alles, was der Künstler modelliert und im finalen Spiel landen soll. Bei einer CD mit einer Speicherkapazität von 650 MB waren diese Daten gut unterzubringen. Im Laufe der Zeit wurden diese Spiele jedoch größer. Der Ausbau von Internetzugängen hat diesem Problem Abhilfe geschaffen.

Mit dem technischen Fortschritt sind auch realistischere Darstellungen in Videospielen möglich. Um die Realität bestmöglich darzustellen, werden Modelle stetig detaillericher, die den steigenden Anforderungen der Hardware gerecht werden. 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 Animationsfilme und Videospielbranche hat die Dreiecksanzahl von 3D-Modellen in die Höhe schießen lassen. Neben den komplexer werdenden Dreiecksnetzen in der Videospielindustrie werden zudem noch hochauflösendere Texturen verwendet. Mit der zunehmenden Komplexität steigt auch der Speicherbedarf.

Aber nicht nur der Speicheraufwand ist ein Problem. Um die Lade- und Renderzeit bei Videospielen zu verkürzen, sind komprimierte Dreiecksnetz, die schnell in den GPU-RAM geladen werden können, um dort dekomprimiert zu werden, essenziell wichtig.

1.3 Ziel der Arbeit

Wie man anhand der Geschichte sieht, war die Datenkompression in ihrer frühen Zeit ein wichtiges Werkzeug, um Informationen weiterzugeben. Diese hat im Laufe der Zeit jedoch immer mehr an Bedeutung für den Alltag gewonnen. Die rasant fortschreitende Digitalisierung führte zur Entwicklung von verbesserten Speichermedien und Leitungen, die höhere Bandbreiten ermöglichen. Die Unterhaltungsbranche hat sehr von der Entwicklung profitiert, jedoch sollen auch Kunden, die keinen guten Internetanschluss besitzen, versorgt werden. Kompressionsalgorithmen sorgen für eine schnellere und zuverlässige Bereitstellung des Services. Aber nicht nur in der Bereitstellung ihres Streaming-Services haben Kompressionsmethoden eine Relevanz bei

Anbietern wie YouTube, Netflix oder Disney+.

Besonders in westlichen Ländern ist die Firma Walt Disney der wohl bekannteste Herausgeber von Animationsfilmen und -serien. Die Art, wie diese Animationsfilme produziert wurden, änderte sich mit den Möglichkeiten der Technik rapide. Die Filme wurden zunächst in Handarbeit gefertigt. 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 neu zu zeichnen, wurden 3D-Modelle für die Charaktere 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 [Dis21].

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 Auftraggebern ein erstes Bild zur Struktur geben. 3D-Drucker können diese Modelle mit Kunststoff in physischer Form herstellen. In Bereichen, in denen Kunststoff nicht robust genug ist, kann Metall mittels CNC-Fräse zu den Modellen geformt werden.

Eine gängige Repräsentation von 3D-Modellen ist die eines Dreiecksnetzes. Mittels Scans von Formen des echten Lebens wie Statuen, Gebäude und Menschen, können zudem sehr große und detaillierte Dreiecksnetze entstehen, die aus sehr vielen Punkten und Dreiecken bestehen. In dieser Arbeit soll der neuartige Kodierungsstandard Brotli-G getestet werden. Ein beliebiges Dreiecksnetz wird dafür zunächst in viele, kleine Dreiecksnetze zerteilt. Die sogenannten *Meshlets* werden in einem eigenen Format gespeichert, das anschließend von Brotli-G auf der CPU komprimiert wird. Die GPU bekommt die Daten, und dekomprimiert die Meshlets, um das gesamte Dreiecksnetz 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 benutzten 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 erreichen des GPU-Buffer bis zum Bildschirm durchlaufen.

2.1 Brotli Kompressionsstandard

Brotli-G ist eine Weiterentwicklung des Brotli Kompressionsstandards, 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. Brotli wurde mit Bedacht auf Kompatibilität mit dem offiziellen Brotli entwickelt. So sollte Brotli auch in der Lage sein, Inhalte, die mit Brotli-G komprimiert wurden, zu entschlüsseln. Zu beachten ist, dass dies nur in dieser Richtung funktioniert, und somit Brotli das Brotli-G 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 (Single Instruction)

MI (Multiple Instruction)

SD (Single Data)

MD (Multiple Data)

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 dem 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 eine langsamere Verarbeitungsgeschwindigkeit gegenüber parallelen Architekturen.

SIMD (Single Instruction, Multiple Data)

Um die Geschwindigkeit zu erhöhen, werden Daten, auf denen dieselbe 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].

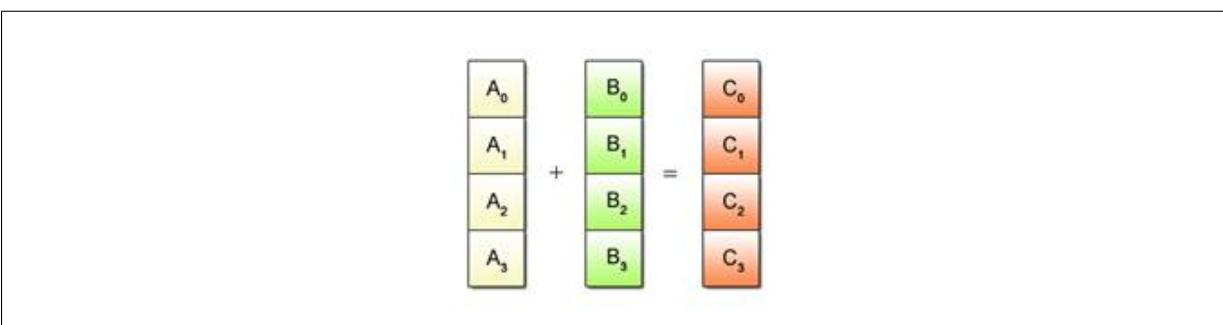


Abb. 2: **Parallele Addition von Daten** Die Abbildung illustriert das SIMD Prinzip anhand eines Beispiels der Vektoraddition. Dabei wird auf verschiedene Daten (die einzelnen Komponenten des Vektors) eine Operation (in diesem Fall eine Addition) ausgeführt. Die Abbildung stammt aus [Son08]

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. Grund dafür war, dass kein Nutzen für diese Architektur gesehen wurde [FR96].

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

GPUs sind Parallelrechner, die auf dem SIMD Prinzip basieren. Die Many-Core Architektur von GPUs hilft dabei, die richtigen Aufgaben schneller und effizienter auszuführen, als es eine CPU könnte. Da GPUs Parallelrechner sind, die auf dem SIMD Prinzip bestehen, sind eben jene Aufgaben gut zugeschnitten, die unter den gleichen Bedingungen mehrfach durchgeführt werden müssen. Wie der Name der *Graphics processing unit* vermuten lässt, wurde diese ursprünglich für die Grafikverarbeitung konzipiert. Hierbei können die vielen Kerne dieselben Operationen auf unterschiedlichen Daten ausführen. Die auf Durchsatz ausgerichteten Rechner verwenden das Programmiermodell *Single Program Multiple Data (SPMD)*. Mit diesem Programmiermodell wird ein Programm (beispielsweise ein HLSL Shader) simultan von mehreren Recheneinheiten ausgeführt. Zusätzlich dazu werden Threads zur weiteren Effizienzsteigerung in SIMD-Gruppen zusammengefasst [YCK14].

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 einer Reihe an programmierbaren (Abb. 3 Grün dargestellt) und fixed-function (Abb. 3 Türkis dargestellt) Stages. Einige dieser Stages sind optional.

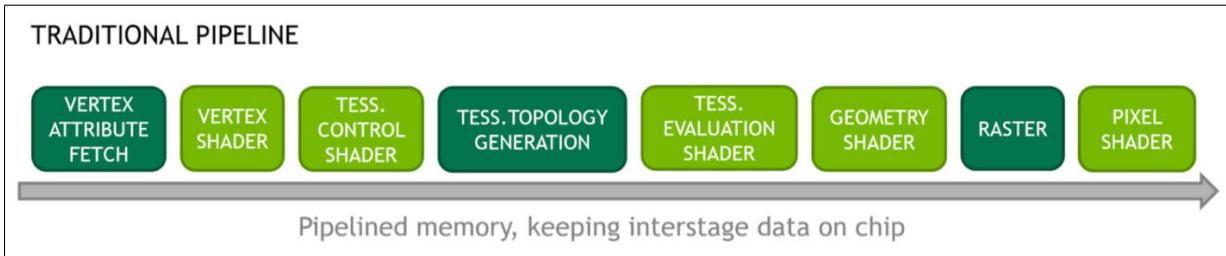


Abb. 3: **traditionelle Rendering Pipeline**

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 Rasterisierer 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 [MW01].

2.3.2 Tessellation Stage

Die Ausgabe Vertices des Vertex Shaders gelangen anschließend in die optionale Tessellation Stage. Der generelle Nutzen ist, einen Patch an Primitiven in wiederum kleinere Primitiven zu verarbeiten. Die Tessellation Stage besteht aus drei Schritten. Darunter ist mit dem *Tessellation Control Shader* (TCS) ein optional programmierbarer Schritt, eine fixed-function Stage 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 zum nächsten Schritt der Pipeline geschickt werden. Im TCS werden der Tessellationsgrad, der Abstand zwischen den Unterteilungen und die gewünschte Topologie festgelegt. Genauer gesagt wird hier gesetzt, wie oft die Primitiven unterteilt werden und welche Topologie 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 wendet der TPG Standard Parameter für die Berechnung an [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 [CLY⁺14].

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 *Fragmenten*. 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 ein Fragment.

2.4 Compute Shader

Der Compute Shader gehört nicht zur traditionellen Grafikpipeline, kann aber mit dieser genutzt werden. Ein Compute Shader dient dazu, alle gewünschten Informationen ohne Einschränkungen 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].

Dementsprechend erwartet der Compute Shader keine spezifischen Daten, wie der Vertex Shader Vertex Attribute erwartet. 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, dass der aktuell aktive/gebundene 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 Threadgroups muss gelten

$$\text{ThreadGroupCountX}, \text{ThreadGroupCountY}, \text{ThreadGroupCountZ} < 65535$$

Im Compute Shader müssen die Anzahl an Threads pro Threadgroup bestimmt werden. Für die Anzahl der Threads muss gelten

$$\text{numThreadsX}, \text{numThreadsY}, \text{numThreadsZ} \leq 128$$

$$\text{numThreadsX} \cdot \text{numThreadsY} \cdot \text{numThreadsZ} = 1024$$

(Für Compute Shader Version 5_0)

Um das SIMD Konzept und den Kontrollfluss des Compute Shaders zu verstehen, sind zwei Variablen elementar wichtig. SVGroupThreadID und SVGroupID. Die GroupID koordiniert die von der Dispatch Methode definierte Anzahl an Threadgroups.

Die GroupID spannt eine bestimmte Anzahl an Threads auf, die vorher im Compute Shader festgelegt wurde. Jedem dieser Threads einer Threadgroup wird eine GroupThreadID zugewiesen.

2.5 Quantisierung von Gleitkommazahlen

Unter Quantisierung versteht man die Reduktion der Genauigkeit von Signalwerten [Str09]. Sie gehört unter den verlustbehafteten Kompressionsmethoden aus dem Bereich der Datenreduktion. Im Rahmen dieser Arbeit sollen die Vertex Attribute quantisiert werden. Die Vertex Attribute bestehen dabei aus einer Position und einer Normalen, die wiederum aus jeweils 3 Gleitkommazahlen bestehen. Da Gleitkommazahlen im Gegensatz zu ganzen Zahlen aus 3 Komponenten bestehen, muss zunächst geklärt werden, wie diese quantisiert werden können.

Um eine Gleitkommazahl zu konstruieren, benötigt man ein Bit für das Vorzeichen (*Sign Bit*), eine Anzahl an Bits für den Exponenten und Mantisse. In der Programmiersprache C++ sind für einer 32 Bit Gleitkommazahl 8 Bit für den Exponenten vorgesehen, während die Mantisse aus 23 Bit besteht (Abb. 4). Der Standard IEEE-754 legt diese Spezifikation für Gleitkommazahlen fest. Diese kann jedoch von Compiler zu Compiler abweichen [Mic23b]. Die Mantisse stellt eine Zahl zwischen 1,0 und 2,0 dar. Aus der Mantisse ist der eigentliche Wert einer Gleitkom-

mazahl zu entnehmen. Der Exponent einer Gleitkommazahl ist für die Skalierung der Mantisse zuständig. Die Mantisse wird mit dem Exponenten multipliziert, um den absoluten Wert der Gleitkommazahl zu erhalten. Das Vorzeichenbit entscheidet anschließend, ob die Zahl positiv oder negativ behaftet ist [WK08].

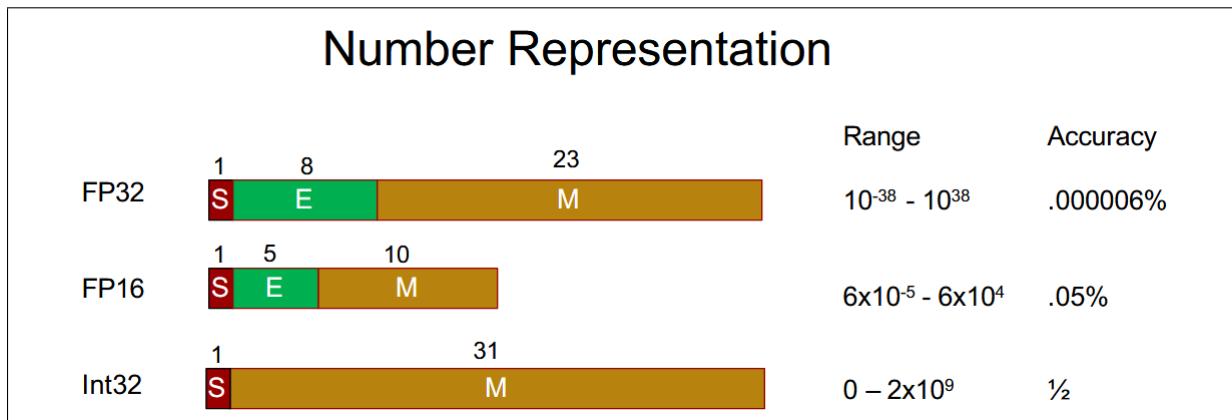


Abb. 4: **Repräsentation einer Zahl** Die Abbildung zeigt die Zusammensetzung von Ganz- und Gleitkommazahlen. Die Abbildung stammt aus dem Foliensatz
<https://media.nips.cc/Conferences/2015/tutorials/slides/Dally-NIPS-Tutorial-2015.pdf>

Das Ziel der Quantisierung ist, eine Gleitkommazahl so zu transformieren, das ihr Wert den originalen Wert so gut es geht widerspiegelt, während der benötigte Speicherplatz verringert wird [Kap10]. Wie in der Abbildung 4 zu sehen ist, wird bei der Repräsentation des *half-precision floating-point formats* der Exponent mit 5 Bit und die Mantisse mit 10 Bit dargestellt.

Der Algorithmus zur Quantisierung von 32 Bit zu 16 Bit Gleitkommazahlen stammt aus dem Meshoptimizer von Zeux [Zeu]. Dazu legt er eine Union an, damit er auf die Eingabe als Ganz- und Gleitkommazahl zugreifen kann. Diese ermöglicht es, auf die Ganzzahl Bitoperationen auszuführen, die durch das Union auch auf der Gleitkommazahl übernommen werden. Das Vorzeichen , der Exponent und der Bias des Exponenten der Zahl werden in den Variablen *s* und *em* und *h* gespeichert. Um Ausnahmefälle zu behandeln, ist eine Überprüfung nach einem Over- oder Underflow notwendig. Zuletzt wird der Wert auf *NaN* (*Not a Number*) gesetzt, falls der Exponent einen gewissen Grenzwert überschreitet.

```

unsigned short meshopt_quantizeHalf(float v)
{
    union { float f; unsigned int ui; } u = {v};
    unsigned int ui = u.ui;

    int s = (ui >> 16) & 0x8000;
    int em = ui & 0x7fffffff;

    int h = (em - (112 << 23) + (1 << 12)) >> 13;

    h = (em < (113 << 23)) ? 0 : h;
    h = (em >= (143 << 23)) ? 0x7c00 : h;
    h = (em > (255 << 23)) ? 0x7e00 : h;

    return (unsigned short)(s | h);
}

```

Code 1: Quantisierung von Float zu Half

2.6 Grundbegriffe der Datenkompression

Das Unterkapitel beschäftigt sich mit den Begriffen der Datenkompression, die im Zuge dieser Arbeit Verwendung finden.

Informationsgehalt

Der Informationsgehalt eines Ereignisses beschreibt, wie viele Informationen beim Auftreten des benannten Ereignisses gewonnen werden können.

$$I(s_i) = \log_2 \frac{1}{p_i}$$

Es sind mehr Informationen zu gewinnen, wenn das auftretende Ereignis eher unwahrscheinlich ist. Was bedeutet, dass der Informationsgehalt eines Ereignisses an Größe gewinnt, je überraschender das Ereignis auftritt. Eine Information wird in der Informationstheorie als *beseitigte Unsicherheit* bezeichnet.

Entropie

Die Entropie einer Datenquelle ist der mittlere Informationsgehalt. Sie berechnet sich aus der Summe der Informationsgehalte der Symbole, multipliziert mit der Auftrittswahrscheinlichkeit des Symbols:

$$H = \sum_{i=1}^K p_i \cdot \log_2 \frac{1}{p_i}.$$

Der Wert der Entropie profitiert von einer Ungleichverteilung der Auftrittswahrscheinlichkeiten. Maximal wird der Wert, wenn eine Gleichverteilung vorherrscht, also wenn die Wahrscheinlichkeiten der Symbole gleich groß sind. Eine geringe Entropie kann erzielt werden, wenn die Auftrittswahrscheinlichkeiten stark voneinander variieren, wie es bei einer Glockenkurve der Fall ist.

Entscheidungsgehalt

Um den Entscheidungsgehalt einer Quelle zu berechnen, wird der Logarithmus Duales auf die Anzahl der Symbole angewendet:

$$H_0 = \log_2(K).$$

Diese Zahl entspricht der Entropie bei einer Gleichverteilung der Symbole.

Kompressionsverhältnis

Aus der Relation der ursprünglichen Daten und den kodierten Daten ergibt sich das Kompressionsverhältnis. Diese gibt ein gutes Bewertungsmaß, wie gut ein Kompressionsalgorithmus für eine gegebene Quelle funktioniert:

$$C_R = \frac{\text{ursprüngliche Datenmenge}}{\text{kodierte Datenmenge}}.$$

Redundanz

Als Redundanz bezeichnet man zusätzlichen Aufwand, der für die Repräsentation einer Information nicht benötigt wird. Um die von Brotli-G verwendeten Algorithmen zu verstehen, muss die Quell- und Codierungsredundanz definiert werden.

Die *Quellredundanz* beschreibt die zusätzlichen Informationen einer Quelle (einer Sammlung an Symbolen), die nicht benötigt werden. Die Quellredundanz ist abhängig von dem Entscheidungsgehalt H_0 des Alphabets und der Quellentropie $H(X)$:

$$R_0 = H_0 - H(X).$$

Unter *Codierungsredundanz* ist die Redundanz zu verstehen, die in den Codewörtern vorherrscht. Die Codierungsredundanz ist die Differenz zwischen der durchschnittlichen Datenmenge (beispielsweise die mittlere Codewortlänge) und der Entropie [Str09].

$$\Delta R(X) = R - H(x)$$

3 Methodik

Die vorliegende Bachelorarbeit beschäftigt sich mit der Dekodierung von Dreiecksnetzen mittels des Kodierungsstandards Brotli-G. Diese Sektion 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, die Dekompressionsgeschwindigkeit und die visuelle Qualität quantitativ ausgewertet werden.

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 Daten für die Meshlets 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 Generierung von Primitiven.

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 zu 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 [Mic21b].

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 5 zu sehen.

Nach diesem Schritt könnten die Daten der UAV von der CPU ausgelesen werden. Mit den Daten können StructuredBuffer gefüllt werden, welche die Daten der Meshlets erhalten. Dieser Schritt ist jedoch als unnötig anzusehen, wenn nicht noch zusätzliche Informationen mit dem dekomprimierten Dreiecksnetz berechnet werden müssen. Der Output Buffer des Brotli-G Dekodierers beinhaltet die benötigten Meshlet Daten, um das Dreiecksnetz zu rekonstruieren. So kann ein GPU-Buffer außerhalb der Brotli-G Klasse angelegt werden, den Brotli-G als

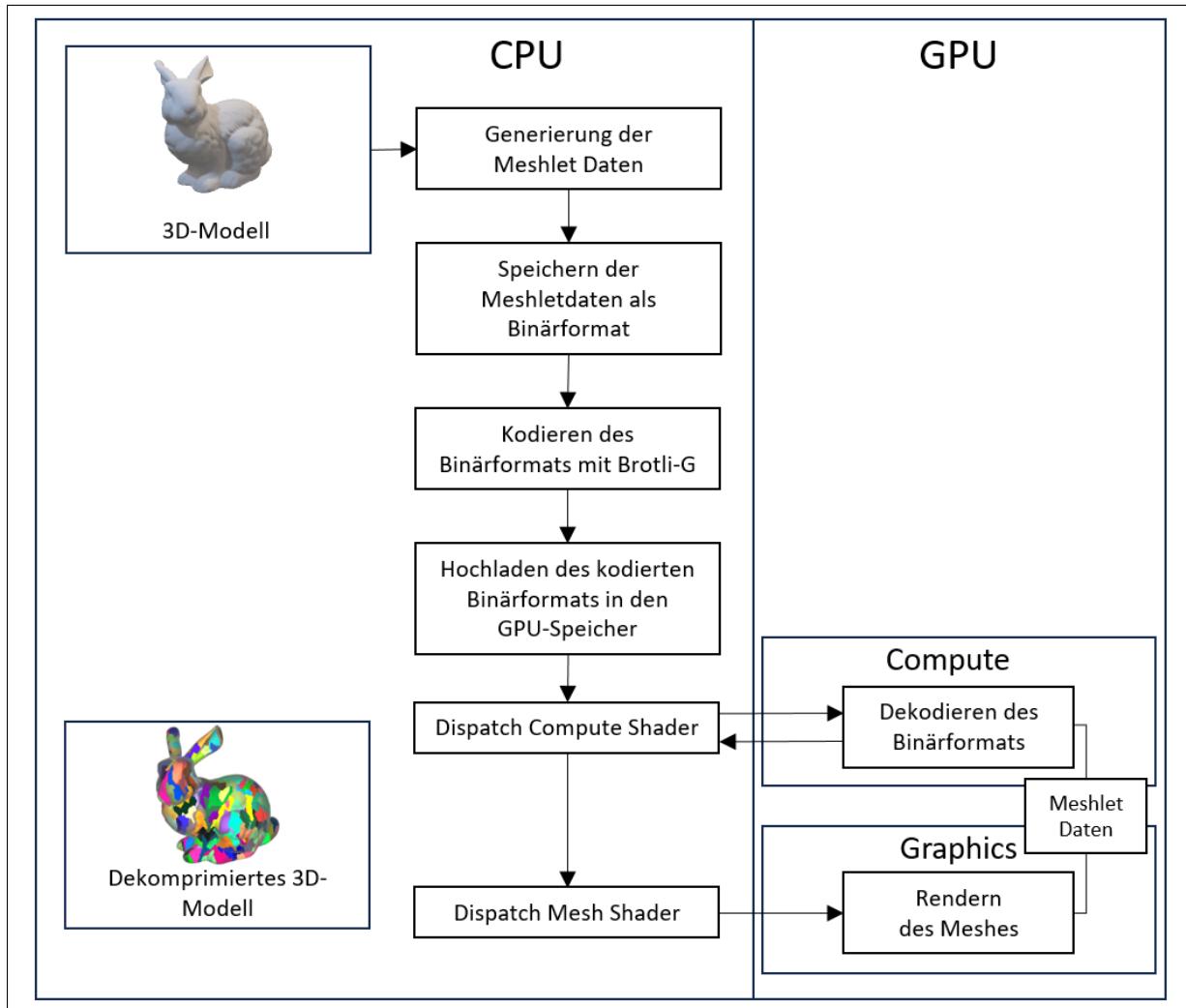


Abb. 5: Flussdiagramm des Ablaufs Abbildung der Dekompressionspipeline.

Output Buffer verwendet. Dieser Buffer und dessen Daten werden nach dem Destruktor von Brotli-Gs Dekodierer nicht freigegeben. Dadurch wird dieser Buffer nicht freigegeben, nachdem der Dekompressionsschritt von Brotli-G abgeschlossen ist. Somit kann der Mesh Shader direkt auf die Daten zugreifen, die nach dem Dekodieren noch im GPU-RAM liegen. Abschließend wird dem Mesh Shader die Speicheradresse des Output Daten Buffer übergeben, aus dem das Dreiecksnetz rekonstruiert werden kann.

4 Mesh Shader

Die Architektur, auf der die 2018 erschienenen 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 Grafikpipeline wird bis heute jedoch noch wenig Beachtung geschenkt. Mit dem Shader Model 6 hat NVIDIA ihre sogenannte „next-generation Grafikpipeline“ 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 Grafikpipeline und das Hauptaugenmerk bei diesem Shader liegt nicht in der Verarbeitung von Geometrie und Topologie [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 Grafikpipeline 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 Die Mesh Shading Pipeline

Die Mesh Shading Pipeline berücksichtigt einige Shader Stages der traditionellen Grafikpipeline aus Kap. 2.3 nicht mehr. Stattdessen sind die Funktionalitäten der verworfenen Stages in den frei Programmierbaren Task- und Mesh Shadern vorhanden.

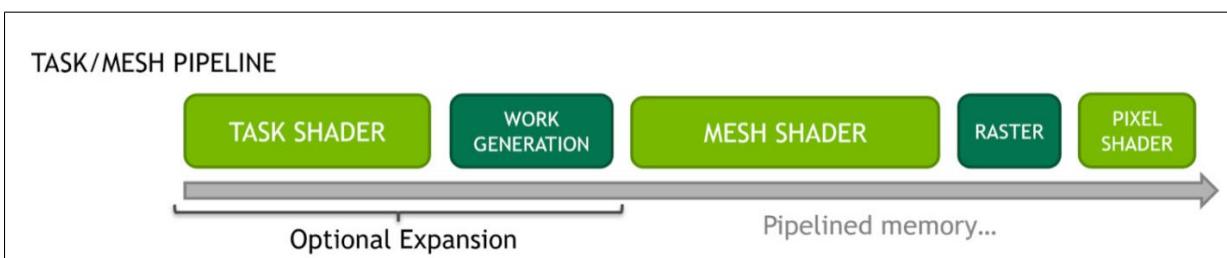


Abb. 6: **Mesh Shading Pipeline** Abbildung der Mesh Shading Pipeline. Die Abbildung ist aus dem NVidia Blogpost [Kub18]

Wie in der Abbildung 11 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 Darstellen des dekomprimierten Dreiecksnetzes genutzt. Zum Rendern müssen jedoch wieder Fragments ausgegeben werden, die in den unveränderte Rasterisierung und Pixel Shader Stages verarbeitet werden. [Kub18]

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. Jedem Meshlet wird eine Workgroup zugeteilt. Jede Workgroup wird wiederum in Threads unterteilt [Kub18].

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].

Wie beim Compute Shader ist der Input des Mesh Shaders anders als bei der traditionellen Grafikpipeline nicht mehr festgelegt. Dadurch kann der Entwickler seine eigenen Implementationen 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 maximale Größe der Meshlets 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. Die 126 ist bewusst keine Zweierpotenz, da 4 Byte für die Anzahl der Primitiven des Meshlets verwendet werden. Zur Vermeidung der Anforderung eines zusätzlichen Speicherblocks werden diese 4 Byte Speicher freigelassen [Kub18].

Arseny Kapoulkine hat verschiedene Meshletgrößen miteinander verglichen. Er ist zu dem Schluss gekommen, dass 64 Vertices und 84 Primitives am effizientesten sind, insbesondere dann, wenn im Task Shader Culling auf die einzelnen Meshlets angewendet 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 4.5). Der letzte Buffer wird in dieser Arbeit als *Meshlet Descriptor* bezeichnet. Der Einfachheit und Übersichtlichkeit halber wird er im Code jedoch einfach als *meshlets* implementiert. Der Meshlet Buffer setzt sich aus folgenden Elementen zusammen [BFB23].

- 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

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 2 wird die Variable `vertexIndex` mittels der `GetVertexIndex` Methode gesetzt. In der Methode wird der lokale Vertex Index mittels

$$\text{localVertex} = \text{VertexOffset} + \text{localIndex}$$

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 Informationen aus 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 anstelle 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:

$$\text{localIndex} = \text{PrimitiveOffset} + (\text{localIndex} \cdot \text{indicesPerTriangle})$$

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 $\hat{V} = 128$ und $\hat{I} = 256$ höchstens 8 Bit für die Repräsentation eines Index benötigt. Diese Auffälligkeit kann sich zunutze gemacht werden,

indem eine Primitive bzw. drei Indizes in ein 4-Byte Integer verpackt werden. Dadurch kann 66 % des Speicherbedarfs für den Index-Buffer gespart werden.

Mit den Informationen der originalen Vertex Daten 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 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 2 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 Queere 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 Primitiven festgelegt. Welchen fundamentalen Nutzen der Meshlet Descriptor für einen Mesh Shader hat, kann man hier erkennen. Die aktuellen Gruppe 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 (drawQuantizedVertices)
    {
        if (localIndex < m.VertCount)
        {
            verts[localIndex] = GetQuantizedVertex(meshletIndex,
                localIndex);
        }
    }
    else
    {
        if (localIndex < m.VertCount)
        {
            verts[localIndex] = GetVertex(meshletIndex, localIndex);
        }
    }
}
```

Code 2: Main Methode des Mesh Shaders

4.5 Lokale Vertex-Buffer

Der Mesh Shader im Codeabschnitt. 2 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 Renderns 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. 7 illustriert ist.



Abb. 7: 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. Dadurch befinden sich zwar duplizierte Vertices im Vertex-Buffer, der Buffer mit den *Unique Vertex Indices* wird jedoch nicht mehr benötigt. Darauf folgt noch, dass der zusätzliche Elementzugriff des wegfallenden Buffers nicht mehr notwendig

ist, um zu den jeweiligen Meshlet Vertex zu gelangen. So besteht jeder einzelne, große Buffer mit allen Elementen sozusagen aus zusammengesetzten, kleinen Meshlet Buffern.

5 Kompressionsstandard Brotli-G

In vielen Anwendungsfällen wird auf Kombinationen von Kompressionsalgorithmen gesetzt. 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 Metablöcken. 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 gemeinsam 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 präfixfreie Codewörter zu generieren. Jeder Metablock hat dabei seine eigene Huffman-Kodierung. So sind Überschneidungen von Codewörtern in unterschiedlichen Metablöcken möglich, während ein präfixfreier Code in einem Metablock gewährleistet ist. Der LZ77 Algorithmus hat zusätzlich noch die Möglichkeit, in einem zuvor kodierten Metablock nach duplizierten Zeichenketten zu suchen; sollte diese Zeichenkette noch im Schiebefenster liegen [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 verzichten, und das Schiebefenster zu Beginn eines neuen Metablocks zurückgesetzt werden [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. Das Ziel des LZ77 Algorithmus ist, redundante Informationen zusammenzufassen, indem die Position und Lauflänge von bereits bekannten Symbolen ausgenutzt werden, um Daten effizient zu komprimieren [Str09].

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. Vorschaufenster: Ein Buffer fester Größe der Symbole vom aktuell betrachteten Symbol bis zum Ende des Buffers enthält
5. Schiebefenster: Daten- und Vorschaufenster
6. Codewort: Ein Codewort besteht aus dem Offset, der Lauflänge und des zu kodierenden Symbols

Zu Beginn des Algorithmus wird das Datenfenster auf den Start des Eingabestroms gesetzt. Dieses Fenster ist zunächst leer. Das Vorschaufenster 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 [Mic23a].

5.1.1 Kodierung eines Codewortes

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

Datenfenster	Vorschaufenster	restliches Codewort	Kodierung
	laufen	raufen	(0, 0)l
1	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-Kodierung 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 und das letzte Symbol aus der Sequenz kodiert werden.

Aus dem Beispiel geht hervor, dass die Auswahl der Buffergröße gut gewählt werden muss, damit der Algorithmus effektiv verwendet werden kann. Hätte das Datenfenster im Beispiel nur Platz für vier statt fünf Symbole gehabt, wäre die Symbolfolge „aufe“ nicht vollständig kodiert worden. 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 des aktuell zu dekodierenden Symbols, 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	laufend
$(6, 4)n$	aufen	aufenauf
\0	raufen	raufenraufen

Tab. 2: Dekodierung mit dem LZ77-Algorithmus

Der LZ77-Algorithmus ist leicht verständlich und effektiv, was ihn für viele Anwendungen nützlich macht. Seine Weiterentwicklungen heißen LZ78 und LZW, die dem LZ77-Algorithmus in einigen Bereichen technisch überlegen sind, aber auch Probleme mit sich bringen. [CPP15]. Hinzu kommt, dass die Weiterentwicklungen aufgrund von Patenten nicht die gleiche Rolle spielen wie die erste Iteration von Lempel und Ziv. Kombiniert mit anderen Verfahren, wie der Huffman-Kodierung, bildet der LZ77-Algorithmus jedoch die Grundlage für viele leistungsfähige Kompressionsstandards, die heute in vielen Anwendungen zu finden sind.

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 dieselbe 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].

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 die 1 erhält oder umgekehrt, ist unerheblich und kann je nach Implementierung variieren. Wichtig ist nur, dass dies im gesamten Baum konsistent durchgeführt 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 [Mof19].

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 Beispielalphabet 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. 8 zu sehen.

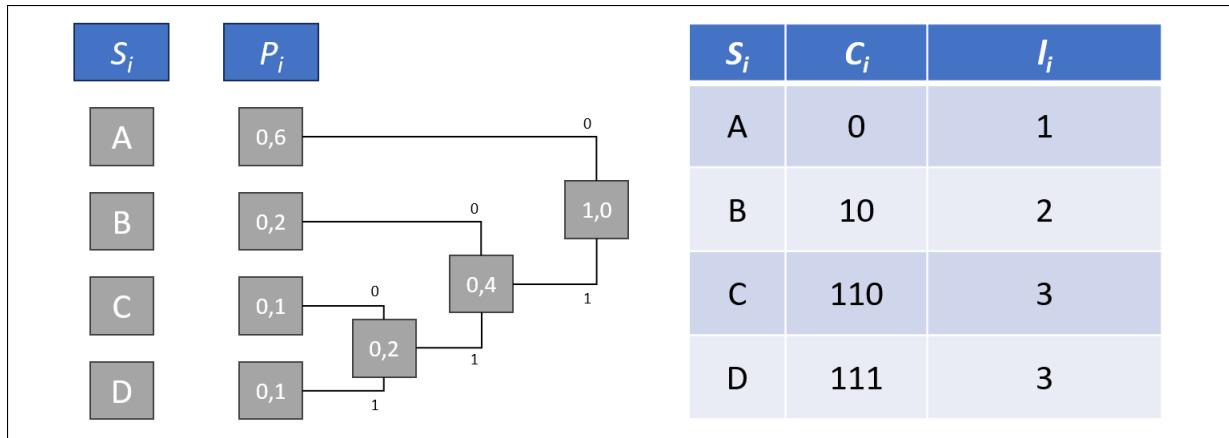


Abb. 8: **Erstellen eines Huffman Codes** Die Abbildung zeigt die Konstruktion eines Binärbaums und der daraus resultierenden Codewörter für die Symbole

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 auf Kap. 2.6 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 [Str09]. So ergibt sich ein Kompressionsverhältnis von:

$$C_R = \frac{[\log_2(N)]}{\sum_{i=1}^n p_i \cdot l_i}$$

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

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 in diesem Fall eine Quelle, wie beispielsweise ein Alphabet betrachtet, ist von der Redundanz einer Quelle oder der *Quellredundanz* die Rede 2.6.

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

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

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

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].

6 Ergebnisse

Dieses Kapitel widmet sich den Ergebnissen dieser Arbeit. Im Kap. ?? 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 zwangsläufig 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. 9: **Der verwendete Datensatz** Die Abbildung zeigt alle Dreiecksnetze mit der Anzahl an Geometrie und Topologie.

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

6.1 Die Ergebnisse 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.

Fandisk			
	Originalgröße	0,366	MB
	Komprimierte Größe	0,160	MB
	Kompressionsverhältnis	2,29	
	CPU	GPU	
Dekompressionszeit	6	3	ms
Bandbreite	0,025	0,054	GiB /s
Vertices	8.675		
Indices	39.040		
Meshlets	136		

Abb. 10: **Fandisk Kompressionsergebnis** Das kleinste Dreiecksnetz aus dem Datensatz, und seine Ergebnisse

Um die niedrige Bandbreite zu analysieren, können die Ergebnisse des Welsh Dragons betrachtet werden. Dieser besteht aus sehr viel mehr Geometrie und Topologie, was sich in der Größe des Modells widerspiegelt. Der Welsh Dragon hat eine Originalgröße von 62.406.096 Bytes, die auf 31.716.764 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 Anbetracht der komprimierten Größe der beiden Dreiecksnetze fällt auf, das der Welsh Dragon sehr viel mehr Daten benötigt. Während

die Dekompression des Welsh Dragons etwas mehr als das Zweifache an Zeit verglichen mit 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.

Welsh Dragon	
	Originalgröße
	62,406 MB
	Komprimierte Größe
	31,717 MB
	Kompressionsverhältnis
	1,97
Vertices 1.473.246	CPU
Indices 6.669.964	GPU
Meshlets 23.021	ms

Abb. 11: **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 Bits mit geringer Wertigkeit, die nur wenig zur Struktur des 3D-Modells beitragen, loszuwerden. Die hinteren 16 Bit einer 32 Bit Gleitkommazahl beansprucht genau soviel Speicher wie die vorderen 16 Bit. Während die vorderen Bits jedoch die grobe Position des Vertex, oder auch die Richtung der Normalen, sind die Bits mit geringerer Wertigkeit für sehr feine Details wichtig. Für ein visuell ansprechendes Dreieck sind 16 Bit pro Komponente ausreichend, außer in Bereichen, in denen diese feine Granularität sehr wichtig ist, wie z.B. bei medizinischen Anwendungen.

Um die Qualität zu visualisieren, sieht man in Abb. 12 das Stanford Bunny einmal mit 16, und einmal mit 32 Bit Vertex Attributen.

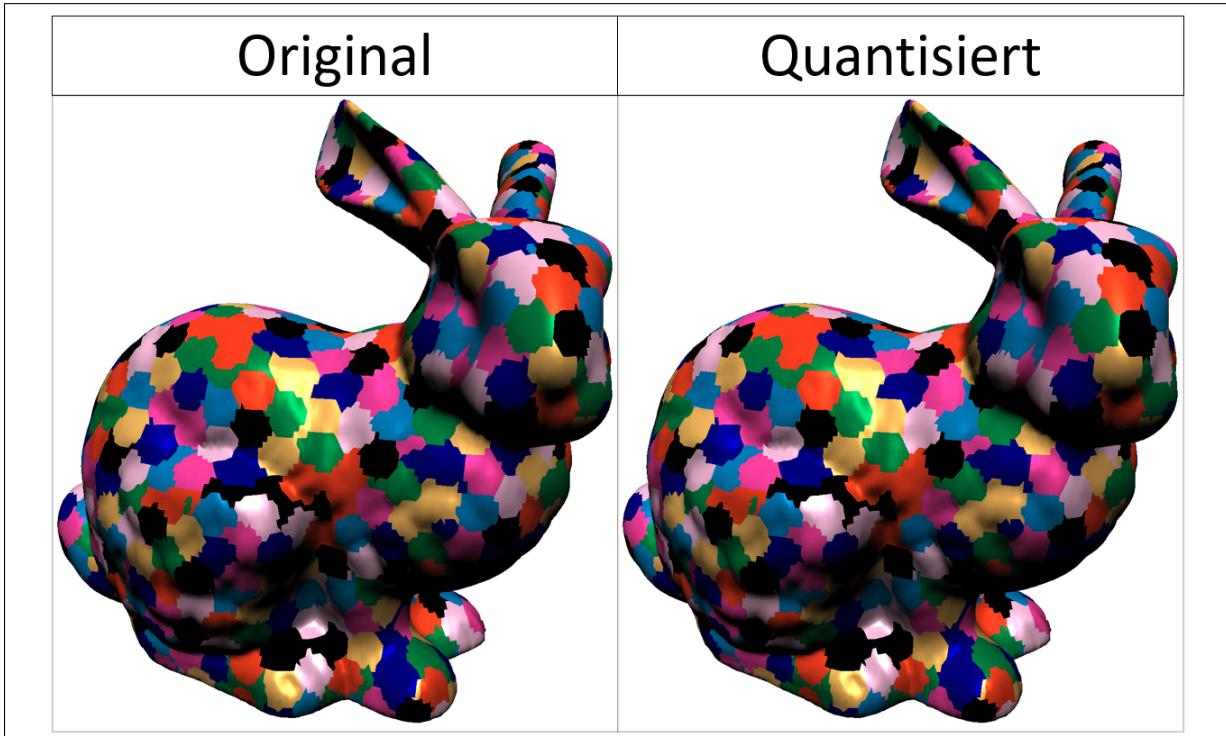


Abb. 12: **Quantisiertes Stanford Bunny** Die Abbildung zeigt eine Gegenüberstellung des Bunny mit 32 Bit Vertex Attributen und dem Bunny mit den quantisierten 16 Bit Vertex Attributen

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 so viel 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, das 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, das die quantisierten Dreiecksnetze nicht nur weniger Speicherplatz beanspruchen. Das Kompressionsverhältnis ist im Durchschnitt des Datensatzes bei den quantisierten Dreiecksnetzen um 18,82 %.

Welsh Dragon Quantized			
			
Originalgröße	44,727	MB	
Komprimierte Größe	17,261	MB	
Kompressionsverhältnis	2,59		
	CPU	GPU	
Dekompressionszeit	285	6	ms
Bandbreite	0,056	2,706	GiB /s
Vertices	1.473.246		
Indices	6.669.964		
Meshlets	23.021		

Abb. 13: **Welsh Dragon Kompressionsergebnis quantisiert** Die Ergebnisse des quantisierten Welsh Dragon

6.3 Die Resultate von Michelangelos David

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 Binärobject 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 $4,682 \text{ GiB/s}$ erreicht wurde.

Auch bei diesem Dreiecksnetz wird zusätzlich noch ein Ergebnis mit quantisierten Vertex Attributen durchgeführt. Die Quantisierung reduziert die Originalgröße des Davids um etwa 200 MB. Nachdem das quantisierte Dreiecksnetz von Brotli-G komprimiert wurde, sind von den

David			
			
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
Vertices	17.963.799		
Indices	68.000.204		
Meshlets	280.706		

Abb. 14: **David Kompressionsergebnis** Ein Teil von dem großen Dreiecksnetz des Davids

544.001.528 Bytes noch 115.287.560 Bytes übrig. Daraus ergibt sich das beste Kompressionsverhältnis in dieser Arbeit mit 4,72. Dafür ist die Bandbreite wiederum etwas schlechter gegenüber dem David mit 32 Bit Gleitkommazahlen für die Vertex-Attribute. Aus einer Dekompressionszeit von 37 ms und der Größe von 115,29 MB ergibt sich dafür eine Bandbreite von 2,983 GiB/s.

Anhand der Ergebnisse scheint der Scan des Davids von Michelangelos sehr detailliert zu sein. Das bedeutet, dass die kleinen Details des Davids von der Quantisierung entfernt wurden, sodass Brotli-G die Daten schöner komprimieren konnte. Hier besteht eine sehr große Differenz der Kompressionsverhältnisse von Originalwerk und dem quantisierten Dreiecksnetz. Während das originale Dreiecksnetz schon ein gutes Kompressionsverhältnis von 2,32 aufweist, ist das Ergebnis des quantisierten Dreiecksnetzes mit 4,72 überragend.

David Quantized															
															
<table><tbody><tr><td>Originalgröße</td><td>544,00</td><td>MB</td><td></td></tr><tr><td>Komprimierte Größe</td><td>115,29</td><td>MB</td><td></td></tr><tr><td>Kompressionsverhältnis</td><td>4,72</td><td></td><td></td></tr></tbody></table>				Originalgröße	544,00	MB		Komprimierte Größe	115,29	MB		Kompressionsverhältnis	4,72		
Originalgröße	544,00	MB													
Komprimierte Größe	115,29	MB													
Kompressionsverhältnis	4,72														
<table><tbody><tr><td>Dekompressionszeit</td><td>5.842</td><td>37</td><td>ms</td></tr><tr><td>Bandbreite</td><td>0,018</td><td>2,893</td><td>GiB /s</td></tr></tbody></table>				Dekompressionszeit	5.842	37	ms	Bandbreite	0,018	2,893	GiB /s				
Dekompressionszeit	5.842	37	ms												
Bandbreite	0,018	2,893	GiB /s												
Vertices	17.963.799														
Indices	68.000.204														
Meshlets	280.706														

Abb. 15: **David Kompressionsergebnis quantisiert** Ein Teil von dem großen Dreiecksnetz des Davids mit quantisierten Vertex Attributen

7 Fazit

Auf den ersten Blick scheint die Bandbreite mit der Größe der komprimierten Daten zu korrelieren. Wie in den Ergebnissen zu sehen, bietet die Quantisierung dazu noch eine nicht zu verachtende Reduktion der Speichergröße. Die Kombination aus Quantisierung und der anschließenden Kompression mit Brotli-G scheint dazu noch bessere Kompressionsverhältnisse zu liefern.

Die Ergebnisse aus Kap. 6 sollen zuletzt bewertet und analysiert werden. Die Bewertungsmaße liegen bei den Kompressionsverhältnissen und der Bandbreite.

7.1 Analyse der Dekompressionszeit und Bandbreite

Eine Möglichkeit für die geringe Bandbreite scheint auf den ersten Blick die Größe der komprimierten Daten zu sein. Auffällig ist nämlich die geringe Bandbreite bei kleinen Dreiecksnetzen wie der Fandisk. Mit einer komprimierten Größe von 0,160 MB benötigt diese am wenigsten Speicherplatz, und erreicht eine Bandbreite von $0,054 \text{ GiB/s}$. Weitere Beispiele sind das Bunny mit einer Größe von $1,067 \text{ MB}$ und einer Bandbreite von $0,503 \text{ GiB/s}$, der Dinosaur mit 1,713 MB und einer Bandbreite von $0,825 \text{ GiB/s}$ und der Rockerarm mit 1,139 MB und einer Bandbreite von $0,399 \text{ GiB/s}$. Es scheint dabei zudem eine untere Grenze der Dekompressionszeit zu geben. Diese liegt bei den sehr kleinen Dreiecksnetzen bei $2 \text{ ms} - 3 \text{ ms}$. Bei Betrachtung der mittelgroßen Dreiecksnetze des Datensatzes scheint die Dekompressionszeit jedoch nicht wie erwartet zuzunehmen. Die Dekompressionszeit der Igea, des Armadillo und des Angels beträgt ebenfalls jeweils 3 ms . Die Größe der komprimierten Dreiecksnetze liegt bei $4,033 \text{ MB}$, $4,486 \text{ MB}$ und $7,239 \text{ MB}$. So sind diese ca. 3 - 4 mal so groß wie die kleinen Dreiecksnetze. Bei einer höheren Dateigröße und gleichbleibender Dekompressionszeit steigt natürlich die Bandbreite. Die Bandbreite der Igea liegt bei $1,473 \text{ GiB/s}$, des Armadillos bei $1,415 \text{ GiB/s}$ und der des Angels bei $2,346 \text{ GiB/s}$. Die Ergebnisse sind zwar besser, aber dennoch noch nicht auf einem hohen Niveau.

Eine mögliche Erklärung für diese untere Schranke kann die Zeitrechnung des Brotli-G Dekodierers sein. Die Zeitrechnung des Brotli-G Dekodierers misst nicht ausschließlich die Dauer des Compute Shaders. Der Umstand einer unteren Schranke bei 2 ms kann am Overhead der API-Aufrufe von DirectX12 liegen. Die Zeitrechnung beginnt demnach bei dem Aufruf der decodeGPU Methode und endet, nachdem der Destruktor der Klasse das Objekt zerstört, das den Compute Shader mit Daten gefüttert und ausgeführt hat.

Die Ergebnisse der zwei größten Dreiecksnetze des Datensatzes scheinen die These zu stützen, dass die kleinen Dreiecksnetze nicht effektiv von Brotli-G verarbeitet werden oder ein zu großer prozentualer Anteil an Overhead das Ergebnis verfälscht. Die Hand erreicht eine Bandbreite von $2,218 \text{ GiB/s}$ bei einer komprimierten Größe von $8,195 \text{ MB}$. Bezuglich der Bandbreite erreicht der Welsh Dragon $4,192 \text{ GiB/s}$, mit einer komprimierten Größe von $31,717 \text{ MB}$. Auch hier scheint die Korrelation von komprimierter Größe und Bandbreite weiterzubestehen.

7.2 Analyse der Kompressionsverhältnisse

Die Kompressionsverhältnisse, die nur mittels Brotli-G erzielt wurden, sind gut. Bei dem Datensatz ergibt sich ein durchschnittliches Kompressionsverhältnis von $1,99$ über alle Dreiecksnetze verteilt. Das größte Kompressionsverhältnis verzeichnet die Fandisk mit $2,29$, während das Horse das geringste Kompressionsverhältnis von $1,84$ besitzt. Bei den Kompressionsverhältnissen gibt es keine Überraschungen oder Auffälligkeiten wie bei der Bandbreite.

7.3 Analyse der quantisierten Dreiecksnetze

Der Großteil der Daten eines Dreiecksnetzes besteht aus seiner Geometrie [JBG17]. Bei Betrachtung der Komponenten ist zwar erkennbar, dass es wesentlich mehr Dreiecke, und somit auch mehr Indizes, als Vertices gibt. Wenn als Topologie eine *Triangle List* verwendet wird, gelangt man auf ein Verhältnis von

$$V : T = 1 : 2$$

Da ein Dreieck aus 3 Indizes besteht, enthält ein Dreiecksnetz von dieser Topologie in etwa 6 Indizes pro Vertex (Euler's Formel) [Eng11]. Angenommen, ein Vertex bestehe lediglich aus Positionen und Normalen, wie es bei dem verwendeten Datensatz dieser Arbeit der Fall ist. So enthält ein Vertex ohne Kompression der Daten

$$3 \cdot 4 \text{ Byte} = 12 \text{ Byte}$$

Gleitkommazahlen für die Position im 3-dimensionalen Raum. Und nochmal so viele Byte für die Normalen. Da gib dem Vertex eine Größe aus 24 Byte. Ein unkomprimierter Index wird in einem Integer ohne Vorzeichen gespeichert, der 4 Byte beansprucht [Mic21a].

Beachtet man das Verhältnis der Euler Formel bei Dreiecksnetzen, enthalten 6 Indizes mit 192 Bit die gleiche Datenmenge wie ein Vertex mit Position und Normale. Zu beachten ist jedoch, dass neben der Position auch Texturkoordinaten, Farben, Tangenten und Bitangenten in den Vertex Attributen vorhanden sein können.

Jedenfalls ist erkennbar, dass die Vertices einen maßgeblichen Einfluss auf die Datengröße

nehmen. Bei jedem einzelnen Dreiecksnetz des Datensatzes ist eine Reduktion der Daten von 28,4 % zu sehen. Die Quantisierung mindert jedoch nicht nur die originalen Eingabedaten. Vergleicht man die von Brotli-G komprimierten Originaldaten und die von Brotli-G komprimierten Daten seines quantisierten Gegenübers, so ist eine Reduktion der komprimierten Daten von durchschnittlich 39,71 % möglich. Die meisten Dreiecksnetze sind in einem Bereich von 38 % - 42 %. Die Ausreißer liegen jedoch nicht weit entfernt von diesem Bereich, mit dem Armadillo der mit 32,6 % am schlechtesten abgeschnitten hat, während der Welsh Dragon mit 45,6 % die größte Reduktion der Originaldaten hat. Diese Diskrepanz der Prozente ist gut erkennbar, wenn die Kompressionsverhältnisse von quantisierten und nicht quantisierten Dreiecksnetzen betrachtet werden. Hier wird schnell klar, das der Armadillo mit 6,42 % den geringsten Zuwachs der Kompressionsverhältnisse erlangt hat, während der Welsh Dragon mit 32,5 % den besten Zuwachs der Kompressionsverhältnisse verzeichnet. Diese komprimierten Daten können direkt aus einem Speichermedium in den GPU-RAM hochgeladen werden, wo sie vom Brotli-G Compute Shader dekomprimiert und vom Mesh Shader gerendert werden.

Das verbesserte Kompressionsverhältnis stammt von dem losgewordenen Rauschen der Originaldaten. Die Daten des Dreiecksnetzes werden für gewöhnlich in einer Genauigkeit repräsentiert, die für die Darstellung einfacher 3D-Modelle nicht benötigt wird. Wenn diese richtig quantisiert werden, ist der Verlust an visueller Qualität meist so gering, das diese kaum auffällt. Die Fließkommazahl ist somit frei von Signalrauschen, das sehr schwer zu kodieren ist, da diese Zeichenfolgen oft zufällig oder willkürlich sind.

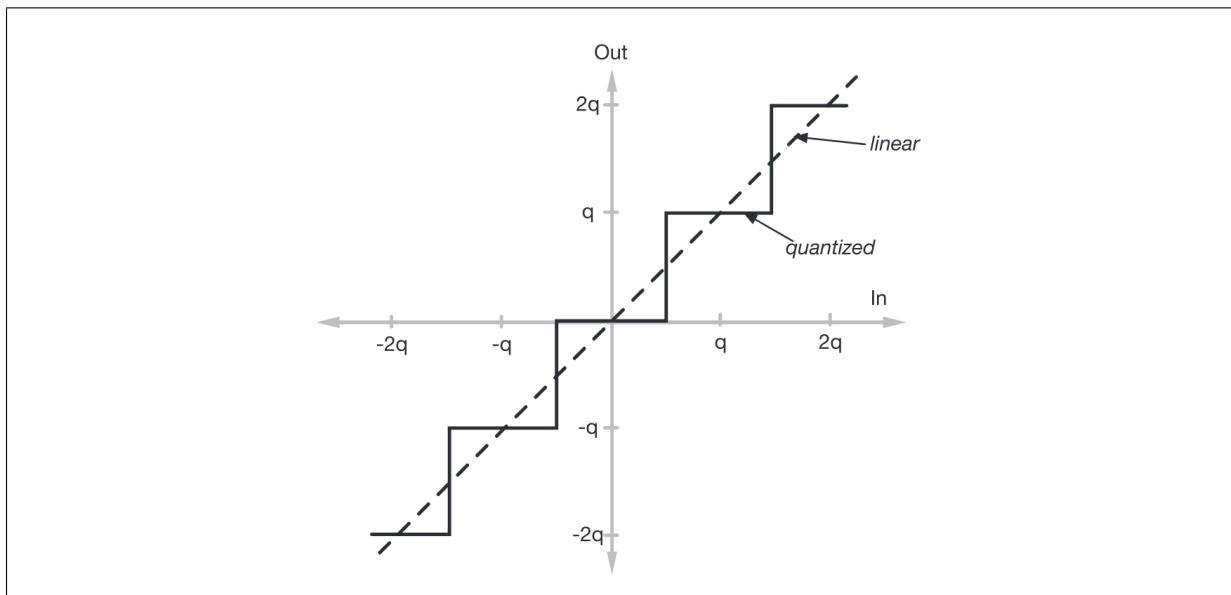


Abb. 16: Mid-Tread Quantisierer Die Abbildung zeigt eine Gegenüberstellung von den möglichen Werten vor der Quantisierung (gestrichelte Linie) und den möglichen Werten nach der Quantisierung (durchgezogene Linie). Die Abbildung stammt aus [Ben20]

7.4 Ausblick für spätere Arbeiten

Wie zu sehen ist, ergeben sich durch die Komprimierung mittels Brotli-G gute Kompressionsverhältnisse. Die Auswertung der Ergebnisse zeigt auch, dass weitere Verfahren zur Datenkompression nicht nur zu einer direkten Datenreduktion führen, sondern auch die Ergebnisse von Brotli-G verbessern können. In dieser Arbeit wurde der Einsatz von der Quantisierung verwendet. Weitere Verfahren, die in Kombination mit Brotli-G getestet werden können, sind die *prädiktive Kodierung*, *Triangle Traversal Encoding* oder *Connectivity Compression* [JBG17]. Da sich die Indizes lediglich in einem bestimmten kleinen Intervall bei der Verwendung von Meshlets befinden, würde sich zusätzlich anbieten, diese nicht einzeln in einer 4 Byte Datenstruktur wie einem Integer zu speichern, sondern jeweils drei Indizes (bei einer Triangle List eine Primitive) in einen Integer zu packen.

Nach einem Blogpost auf GPUOpen von AMD wurde die Version 1.1 von Brotli-G mit dem Compressorator Version 4.5 veröffentlicht. Die Tests die in dem Blogpost behandelt werden, richten sich zwar an Texturen, die Ergebnisse von der neuen Version von Brotli-G sind dennoch sehenswert. Im Vergleich zur alten Version reduziert der Compressorator mit der neuen Brotli-G Version die komprimierten Texturen im Durchschnitt um weitere 10 % - 15 %, in besonders guten Fällen sogar um 20 % [Lev24]. Die Brotli-G Version 1.1 wurde leider erst gegen Ende dieser Arbeit veröffentlicht, wodurch die vorliegenden Ergebnisse auf der Version 1.0 bestehen. Die besseren Ergebnisse des Compressorators lassen auf zukünftig bessere Resultate und vor allem auf eine effektivere Ausnutzung der Bandbreite des PCIe-Busses hoffen. Die Ergebnisse dieser Arbeit in der Hinsicht der Dekompressionszeit und Bandbreitenausnutzung sind selbst bei den größeren Dreiecksnetzen weit von gewünschten Werten entfernt. Für die Versuche wurde die Grafikkarte *Radeon RX 6650 XT* von AMD verwendet. Diese soll laut Datenbankeintrag eine maximale Bandbreite von 280 GB/s, also zum Vergleich umgerechnet ungefähr 260 GiB/s erreichen. Es ist zwar unwahrscheinlich, an dieses Maximum zu gelangen, dennoch wäre eine bessere Ausnutzung der Bandbreite wünschenswert.

		Ratio	H100 PCIe	
			Compression Throughput (GB/s)	Decompression Throughput (GB/s)
Graphics: Geometry data				
	lz4	1.40	14.75	51.77
	snappy	1.46	14.91	43.78
	cascaded	1.45	171.27	376.77
	gdeflate-high-throughput	1.86	11.02	69.04
	gdeflate-high-compression	2.17	0.37	53.52
	gdeflate-entropy-only	1.58	104.85	72.34
	bitcomp-default	1.85	536.26	371.71
	bitcomp-sparse	1.46	708.52	749.11
	deflate	1.87	10.46	15.63
	ans	1.55	271.12	372.29
	zstd	2.18	8.98	38.77

Abb. 17: NVIDIA's nvCOMP Testergebnisse In der Abbildung sind die Ergebnisse der NVIDIA Bibliothek nvCOMP zu sehen, die verlustfreie Datenkompression und Dekompression unter Verwendung einer GPU ausführt. Die Abbildung ist von NVIDIA's öffentlicher Developer Website. <https://developer.nvidia.com/nvcomp>

In der Abbildung. 17 ist zu sehen, welche Kompressionsverhältnisse unterschiedliche Algorithmen der nvCOMP Bibliothek. Neben dieser sind die Bandbreiten der Kompression und Dekompression. Das Ziel ist es nicht, die Bandbreiten dieser Arbeit mit den Ergebnissen von nvCOMP zu vergleichen. Dabei spielt auch nicht nur der verwendete Datensatz eine Rolle. Die verwendete GPU aus Abbildung 17 ist nicht für Privatpersonen gebaut worden. Vor allem ist die *Tensor Core GPU* H100 für Unternehmen gedacht, die GPUs für aufwendige Rechenarbeiten benötigen. Dementsprechend erzielt diese sehr viel bessere Ergebnisse, als die für Privatpersonen geeignete RX 6650 XT. Dennoch kann aus dieser Abbildung entnommen werden, dass eine sehr viel höhere Bandbreite erzielt werden kann, als es in dieser Arbeit der Fall war.

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/>. – besucht am 27.11.2023
- [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)
- [Ben20] BENNETT, Christopher L.: *Digital Audio Theory: A Practical Guide*. 2020 <libgen.li/file.php?md5=3498477d0a02ff6d40297c0cf73bac79>. – ISBN 2020031086; 9782020031080; 9780367276553; 0367276550; 9780367276539; 0367276534; 9780429297144; 0429297149
- [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)
- [CLY⁺14] CHANG, Hsu-Huai ; LAI, Yu-Chi ; YAO, Chin-Yuan ; HUA, Kai-Lung ; NIU, Yuzhen ; LIU, Feng: Geometry-shader-based real-time voxelization and applications. In: *The Visual Computer* 30 (2014), S. 327–340
- [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
- [Dis21] The History of the Evolving Animation Styles of Disney. (2021). <https://www.thedisneyclassics.com/blog/animation-styles>. – besucht am 01.04.2024

- [Eng11] ENGSTAD, Pål-Kristian: Ratio of vertexes, edges and triangles in a Mesh. (2011). https://jahej.com/alt/2011_06_15_ratio-of-vertexes-edges-and-triangles-in-a-mesh.html. – besucht am 24.03.2024
- [FR96] FLYNN, Michael J. ; RUDD, Kevin W.: Parallel architectures. In: *ACM computing surveys (CSUR)* 28 (1996), Nr. 1, S. 67–70
- [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/>. – besucht am 10.01.2024
- [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
- [Kap10] KAPOULKINE, Arseny: Quantizing floats. (2010)
- [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>
- [Lev24] LEVESQUE, Denis: Introducing Compressorator v4.5 with up to 20Brotli-G compression. (2024). <https://gpuopen.com/learn/compressorator-v4-5-improved-brotli-g-compression/>. – besucht am 28.03.2024
- [Mic21a] MICROSOFT: Data Type Ranges. (2021). <https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=msvc-170>. – besucht am 29.03.2024

- [Mic21b] MICROSOFT: Typed Unordered Access Views (UAV). In: *Microsoft Documentation* (2021). <https://learn.microsoft.com/en-us/windows/win32/direct3d12/typed-unordered-access-view-loads>. – besucht am 25.01.2024
- [Mic23a] MICROSOFT: LZ77 Compression Algorithm. In: *Microsoft Documentation* (2023). https://learn.microsoft.com/en-usopenspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb. – besucht am 26.01.2024
- [Mic23b] MICROSOFT: Typ "float". (2023). <https://learn.microsoft.com/de-de/cpp/c-language/type-float?view=msvc-170>. – besucht am 29.03.2024
- [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
- [MW01] MAUGHAN, Chris ; WLOKA, Matthias: Vertex shader introduction. In: *NVIDIA Technical Brief* (2001)
- [Ope24] OPENGL: Compute Shader. In: *OpenGL Wiki* ((besucht am 09.02.2024)). https://www.khronos.org/opengl/wiki/Compute_Shader
- [Son08] SONY: Cell Programming Primer. (2008). <http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingPrimer.html>. – besucht am 22.11.2023
- [Str09] STRUTZ, Tilo: *Bilddatenkompression*. Vieweg+Teubner, 2009. <http://dx.doi.org/10.1007/978-3-8348-9986-6>. – ISBN 9783834899866
- [WK08] In: WIDROW, Bernard ; KOLLÁR, István: *Basics of Floating–Point Quantization*. Cambridge University Press, 2008, S. 257–306
- [YCK14] YILMAZER, Ayse ; CHEN, Zhongliang ; KAELI, David: Scalar Waving: Improving the Efficiency of SIMD Execution on GPUs. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, S. 103–112
- [Zeu] ZEUX: meshoptimizer. <https://github.com/zeux/meshoptimizer>

A1 Anhang

Gitea-Seite des Projekts: https://git.hs-coburg.de/jan2118s/Geometry_Decompression

A1.1 Ergebnisse ohne Quantisierung

Angel		
Compression Ratio : 1.85		
• Kompression	• CPU Dekompression	• GPU Dekompression
Input size : 13416096 bytes	Input size : 7238976 bytes	Input size : 7238976 bytes
Output size : 7238976 bytes	Output size : 13416096 bytes	Output size : 13416096 bytes
Processed in : 1,587 s	Processed in : 69 ms	Processed in : 3 ms
Bandwidth : 0.007873 GiB/s	Bandwidth : 0.097708 GiB/s	Bandwidth : 2.346124 GiB/s
• Mesh		
Number of Vertices: 317440		
Number of Indices: 1429540		
Number of Meshlets: 4961		

Armadillo

Compression Ratio : 2.18

• Kompression

Input size : 9777368 bytes
Output size : 4486088 bytes
Processed in : 1,224 s
Bandwidth : 0.007439 GiB/s

• CPU Dekompression

Input size : 4486088 bytes
Output size : 9777368 bytes
Processed in : 54 ms
Bandwidth : 0.077370 GiB/s

• GPU Dekompression

Input size : 4486088 bytes
Output size : 9777368 bytes
Processed in : 3 ms
Bandwidth : 1.415176 GiB/s

• Mesh

Number of Vertices: 231081
Number of Indices: 1043412
Number of Meshlets: 3611



Bunny

Compression Ratio : 1.85

• Kompression

Input size : 1978544 bytes
Output size : 1067388 bytes
Processed in : 255 ms
Bandwidth : 0.007226 GiB/s

• CPU Dekompression

Input size : 1067388 bytes
Output size : 1978544 bytes
Processed in : 18 ms
Bandwidth : 0.055227 GiB/s

• GPU Dekompression

Input size : 1067388 bytes
Output size : 1978544 bytes
Processed in : 2 ms
Bandwidth : 0.503670 GiB/s

• Mesh

Number of Vertices: 46930
Number of Indices: 210120
Number of Meshlets: 734



Dinosaur

Compression Ratio : 1.86

- Kompression

Input size : 3179112 bytes
Output size : 1713272 bytes
Processed in : 412 ms
Bandwidth : 0.007186 GiB/s

- CPU Dekompression

Input size : 1713272 bytes
Output size : 3179112 bytes
Processed in : 28 ms
Bandwidth : 0.056986 GiB/s

- GPU Dekompression

Input size : 1713272 bytes
Output size : 3179112 bytes
Processed in : 2 ms
Bandwidth : 0.824570 GiB/s

- Mesh

Number of Vertices: 75189
Number of Indices: 338944
Number of Meshlets: 1175



Fandisk

Compression Ratio : 2.29

- Kompression

Input size : 366536 bytes
Output size : 160200 bytes
Processed in : 79 ms
Bandwidth : 0.004321 GiB/s

- CPU Dekompression

Input size : 160200 bytes
Output size : 366536 bytes
Processed in : 6 ms
Bandwidth : 0.024866 GiB/s

- GPU Dekompression

Input size : 160200 bytes
Output size : 366536 bytes
Processed in : 3 ms
Bandwidth : 0.053967 GiB/s

- Mesh

Number of Vertices: 8675
Number of Indices: 39040
Number of Meshlets: 136



Hand

Compression Ratio : 2.26

• Kompression

Input size : 18502280 bytes
Output size : 8194940 bytes
Processed in : 2,134 s
Bandwidth : 0.008075 GiB/s

• CPU Dekompression

Input size : 8194940 bytes
Output size : 18502280 bytes
Processed in : 98 ms
Bandwidth : 0.077879 GiB/s

• GPU Dekompression

Input size : 8194940 bytes
Output size : 18502280 bytes
Processed in : 3 ms
Bandwidth : 2.217560 GiB/s

• Mesh

Number of Vertices: 437283
Number of Indices: 1974540
Number of Meshlets: 6833



Horse

Compression Ratio : 1.84

• Kompression

Input size : 2746856 bytes
Output size : 1494476 bytes
Processed in : 369 ms
Bandwidth : 0.006933 GiB/s

• CPU Dekompression

Input size : 1494476 bytes
Output size : 2746856 bytes
Processed in : 23 ms
Bandwidth : 0.060515 GiB/s

• GPU Dekompression

Input size : 1494476 bytes
Output size : 2746856 bytes
Processed in : 2 ms
Bandwidth : 0.581214 GiB/s

• Mesh

Number of Vertices: 65029
Number of Indices: 292472
Number of Meshlets: 1017



Igea

Compression Ratio : 1.89

• Kompression

Input size : 7621696 bytes
Output size : 4033480 bytes
Processed in : 966 ms
Bandwidth : 0.007348 GiB/s

• CPU Dekompression

Input size : 4033480 bytes
Output size : 7621696 bytes
Processed in : 43 ms
Bandwidth : 0.087360 GiB/s

• GPU Dekompression

Input size : 4033480 bytes
Output size : 7621696 bytes
Processed in : 3 ms
Bandwidth : 1.473259 GiB/s

• Mesh

Number of Vertices: 180672
Number of Indices: 810096
Number of Meshlets: 2824



Isis

Compression Ratio : 1.95

• Kompression

Input size : 10611288 bytes
Output size : 5446608 bytes
Processed in : 1,313 s
Bandwidth : 0.007527 GiB/s

• CPU Dekompression

Input size : 5446608 bytes
Output size : 10611288 bytes
Processed in : 63 ms
Bandwidth : 0.080517 GiB/s

• GPU Dekompression

Input size : 5446608 bytes
Output size : 10611288 bytes
Processed in : 3 ms
Bandwidth : 1.677980 GiB/s

• Mesh

Number of Vertices: 251363
Number of Indices: 1128932
Number of Meshlets: 3928



Rockerarm

Compression Ratio : 2.00

• Kompression

Input size : 2278104 bytes
Output size : 1138632 bytes
Processed in : 335 ms
Bandwidth : 0.006333 GiB/s

• CPU Dekompression

Input size : 1138632 bytes
Output size : 2278104 bytes
Processed in : 23 ms
Bandwidth : 0.046106 GiB/s

• GPU Dekompression

Input size : 1138632 bytes
Output size : 2278104 bytes
Processed in : 3 ms
Bandwidth : 0.398647 GiB/s

• Mesh

Number of Vertices: 53967
Number of Indices: 242348
Number of Meshlets: 844



Welsh Dragon

Compression Ratio : 1.97

• Kompression

Input size : 62406096 bytes
Output size : 31716764 bytes
Processed in : 7,464 s
Bandwidth : 0.007787 GiB/s

• CPU Dekompression

Input size : 31716764 bytes
Output size : 62406096 bytes
Processed in : 407 ms
Bandwidth : 0.072576 GiB/s

• GPU Dekompression

Input size : 31716764 bytes
Output size : 62406096 bytes
Processed in : 7 ms
Bandwidth : 4.192385 GiB/s

• Mesh

Number of Vertices: 1473246
Number of Indices: 6669964
Number of Meshlets: 23021



David Gesamt

Compression Ratio : 2.32

• Kompression

Input size : 759567128 bytes
Output size : 326929888 bytes
Processed in : 1,50 min
Bandwidth : 0.006382 GiB/s

• CPU Dekompression

Input size : 326929888 bytes
Output size : 759567128 bytes
Processed in : 17,878 s
Bandwidth : 0.017031 GiB/s

• GPU Dekompression

Input size : 326929888 bytes
Output size : 759567128 bytes
Processed in : 65 ms
Bandwidth : 4.682219 GiB/s

• Mesh

Number of Vertices: 1473246
Number of Indices: 6669964
Number of Meshlets: 23021



A1.2 Ergebnisse mit Quantisierung

Angel Quantized

Compression Ratio : 2.30

• Kompression

Input size : 9606816 bytes

Output size : 4170944 bytes

Processed in : 1,203 s

Bandwidth : 0.007437 GiB/s

• CPU Dekompression

Input size : 4170944 bytes

Output size : 9606816 bytes

Processed in : 67 ms

Bandwidth : 0.057978 GiB/s

• GPU Dekompression

Input size : 4170944 bytes

Output size : 9606816 bytes

Processed in : 3 ms

Bandwidth : 1.326518 GiB/s

• Mesh

Number of Vertices: 317440

Number of Indices: 1429540

Number of Meshlets: 4961



Armadillo Quantized

Compression Ratio : 2.32

- Kompression

Input size : 7004384 bytes
Output size : 3024124 bytes
Processed in : 963 ms
Bandwidth : 0.006774 GiB/s

- CPU Dekompression

Input size : 3024124 bytes
Output size : 7004384 bytes
Processed in : 59 ms
Bandwidth : 0.047736 GiB/s

- GPU Dekompression

Input size : 3024124 bytes
Output size : 7004384 bytes
Processed in : 3 ms
Bandwidth : 0.951875 GiB/s

- Mesh

Number of Vertices: 231081
Number of Indices: 1043412
Number of Meshlets: 3611



Bunny Quantized

Compression Ratio : 2.29

- Kompression

Input size : 1415384 bytes
Output size : 618360 bytes
Processed in : 231 ms
Bandwidth : 0.005706 GiB/s

- CPU Dekompression

Input size : 618360 bytes
Output size : 1415384 bytes
Processed in : 16 ms
Bandwidth : 0.035993 GiB/s

- GPU Dekompression

Input size : 618360 bytes
Output size : 1415384 bytes
Processed in : 2 ms
Bandwidth : 0.266526 GiB/s

- Mesh

Number of Vertices: 46930
Number of Indices: 210120
Number of Meshlets: 734



Dinosaur Quantized

Compression Ratio : 2.23

• Kompression

Input size : 2276832 bytes
Output size : 1020136 bytes
Processed in : 322 ms
Bandwidth : 0.006585 GiB/s

• CPU Dekompression

Input size : 1020136 bytes
Output size : 2276832 bytes
Processed in : 22 ms
Bandwidth : 0.043185 GiB/s

• GPU Dekompression

Input size : 1020136 bytes
Output size : 2276832 bytes
Processed in : 2 ms
Bandwidth : 0.463366 GiB/s

• Mesh

Number of Vertices: 75189
Number of Indices: 338944
Number of Meshlets: 1175



Fandisk Quantized

Compression Ratio : 2.82

• Kompression

Input size : 262424 bytes
Output size : 92972 bytes
Processed in : 78 ms
Bandwidth : 0.003133 GiB/s

• CPU Dekompression

Input size : 92972 bytes
Output size : 262424 bytes
Processed in : 6 ms
Bandwidth : 0.014431 GiB/s

• GPU Dekompression

Input size : 92972 bytes
Output size : 262424 bytes
Processed in : 3 ms
Bandwidth : 0.033982 GiB/s

• Mesh

Number of Vertices: 8675
Number of Indices: 39040
Number of Meshlets: 136



Hand Quantized

Compression Ratio : 2.44

• Kompression

Input size : 13254872 bytes
Output size : 5422692 bytes
Processed in : 1,724 s
Bandwidth : 0.007160 GiB/s

• CPU Dekompression

Input size : 5422692 bytes
Output size : 13254872 bytes
Processed in : 90 ms
Bandwidth : 0.056114 GiB/s

• GPU Dekompression

Input size : 5422692 bytes
Output size : 13254872 bytes
Processed in : 3 ms
Bandwidth : 1.562068 GiB/s

• Mesh

Number of Vertices: 437283
Number of Indices: 1974540
Number of Meshlets: 6833



Horse Quantized

Compression Ratio : 2.25

• Kompression

Input size : 1966496 bytes
Output size : 874252 bytes
Processed in : 315 ms
Bandwidth : 0.005814 GiB/s

• CPU Dekompression

Input size : 874252 bytes
Output size : 1966496 bytes
Processed in : 25 ms
Bandwidth : 0.032568 GiB/s

• GPU Dekompression

Input size : 874252 bytes
Output size : 1966496 bytes
Processed in : 2 ms
Bandwidth : 0.328935 GiB/s

• Mesh

Number of Vertices: 65029
Number of Indices: 292472
Number of Meshlets: 1017



Igea Quantized

Compression Ratio : 2.29

• Kompression

Input size : 5453632 bytes
Output size : 2380876 bytes
Processed in : 731 ms
Bandwidth : 0.006948 GiB/s

• CPU Dekompression

Input size : 2380876 bytes
Output size : 5453632 bytes
Processed in : 43 ms
Bandwidth : 0.051567 GiB/s

• GPU Dekompression

Input size : 2380876 bytes
Output size : 5453632 bytes
Processed in : 3 ms
Bandwidth : 0.793716 GiB/s

• Mesh

Number of Vertices: 180672
Number of Indices: 810096
Number of Meshlets: 2824



Isis Quantized

Compression Ratio : 2.27

• Kompression

Input size : 7594920 bytes
Output size : 3343480 bytes
Processed in : 966 ms
Bandwidth : 0.007322 GiB/s

• CPU Dekompression

Input size : 3343480 bytes
Output size : 7594920 bytes
Processed in : 56 ms
Bandwidth : 0.055605 GiB/s

• GPU Dekompression

Input size : 3343480 bytes
Output size : 7594920 bytes
Processed in : 3 ms
Bandwidth : 1.141201 GiB/s

• Mesh

Number of Vertices: 251363
Number of Indices: 1128932
Number of Meshlets: 3928



Rockerarm Quantized

Compression Ratio : 2.27

• Kompression

Input size : 1630488 bytes
Output size : 719808 bytes
Processed in : 239 ms
Bandwidth : 0.006354 GiB/s

• CPU Dekompression

Input size : 719808 bytes
Output size : 1630488 bytes
Processed in : 16 ms
Bandwidth : 0.041898 GiB/s

• GPU Dekompression

Input size : 719808 bytes
Output size : 1630488 bytes
Processed in : 2 ms
Bandwidth : 0.297176 GiB/s

• Mesh

Number of Vertices: 53967
Number of Indices: 242348
Number of Meshlets: 844



Welsh Dragon Quantized

Compression Ratio : 2.59

• Kompression

Input size : 44727144 bytes
Output size : 17260728 bytes
Processed in : 5,736 s
Bandwidth : 0.007262 GiB/s

• CPU Dekompression

Input size : 17260728 bytes
Output size : 44727144 bytes
Processed in : 285 ms
Bandwidth : 0.056405 GiB/s

• GPU Dekompression

Input size : 17260728 bytes
Output size : 44727144 bytes
Processed in : 6 ms
Bandwidth : 2.706153 GiB/s

• Mesh

Number of Vertices: 1473246
Number of Indices: 6669964
Number of Meshlets: 23021



David Gesamt Quantized

Compression Ratio : 4.72

• Kompression

Input size : 544001528 bytes
Output size : 115287560 bytes
Processed in : 1,33 min
Bandwidth : 0.005424 GiB/s

• CPU Dekompression

Input size : 115287560 bytes
Output size : 544001528 bytes
Processed in : 5,842 s
Bandwidth : 0.018379 GiB/s

• GPU Dekompression

Input size : 115287560 bytes
Output size : 544001528 bytes
Processed in : 37 ms
Bandwidth : 2.893051 GiB/s

• Mesh

Number of Vertices: 1473246
Number of Indices: 6669964
Number of Meshlets: 23021



A1.3 Berechnung der verwendeten Prozentzahlen

Prozentuales Verhältnis Originale/quantisierte Daten

Originaldaten

Dreiecksnetz	Original (in MB)	Quantisiert (in MB)	Datenreduktion
Angel	13,416	9,607	$\frac{(13,416 - 9,607)}{13,416} = 28,4\%$
Armadillo	9,777	7,004	$\frac{(9,777 - 7,004)}{9,777} = 28,4\%$
Bunny	1,978	1,415	$\frac{(1,978 - 1,415)}{1,978} = 28,5\%$
Dinosaur	3,179	2,277	$\frac{(3,179 - 2,277)}{3,179} = 28,4\%$
Fandisk	0,366	0,262	$\frac{(0,366 - 0,262)}{0,366} = 28,4\%$
Hand	18,502	13,255	$\frac{(18,502 - 13,255)}{18,502} = 28,4\%$
Horse	2,747	1,966	$\frac{(2,747 - 1,966)}{2,747} = 28,4\%$
Igea	7,621	5,454	$\frac{(7,621 - 5,454)}{7,621} = 28,4\%$
Isis	10,611	7,595	$\frac{(10,611 - 7,595)}{10,611} = 28,4\%$
Rockerarm	2,278	1,630	$\frac{(2,278 - 1,630)}{2,278} = 28,4\%$
Welsh Dragon	62,406	44,727	$\frac{(62,406 - 44,727)}{62,406} = 28,3\%$
$\varnothing = 28,4\%$			

Prozentuales Verhältnis Originale/quantisierte Daten

Komprimierte Daten

Dreiecksnetz	Original (in MB)	Quantisiert (in MB)	Datenreduktion
Angel	7,239	4,171	$\frac{(7,239 - 4,171)}{7,239} = 42,4\%$
Armadillo	4,486	3,024	$\frac{(4,486 - 3,024)}{4,486} = 32,6\%$
Bunny	1,067	0,618	$\frac{(1,067 - 0,618)}{1,067} = 42,1\%$
Dinosaur	1,713	1,020	$\frac{(1,713 - 1,020)}{1,713} = 40,5\%$
Fandisk	0,160	0,093	$\frac{(0,160 - 0,093)}{0,160} = 41,9\%$
Hand	8,195	5,422	$\frac{(8,195 - 5,422)}{8,195} = 33,8\%$
Horse	1,494	0,874	$\frac{(1,494 - 0,874)}{1,494} = 41,5\%$
Igea	4,033	2,381	$\frac{(4,033 - 2,381)}{4,033} = 41,0\%$
Isis	5,447	3,343	$\frac{(5,447 - 3,343)}{5,447} = 38,6\%$
Rockerarm	1,139	0,720	$\frac{(1,139 - 0,720)}{1,139} = 36,8\%$
Welsh Dragon	31,717	17,261	$\frac{(31,717 - 17,261)}{31,717} = 45,6\%$
$\varnothing = 39,71\%$			

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