HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg
Fakultät Elektrotechnik und Informatik

Studiengang: Informatik

Bachelorarbeit

Geometriedekompression von Dreiecksnetzen auf der GPU

Janek Foote

Abgabe des Arbeit: 13.03.2024

Betreut durch:

Quirin Meyer, Hochschule Coburg

Inhaltsverzeichnis

Abbildungsverzeichnis

Codebeispielverzeichnis

1 Einführung

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

1.1 Geschichtliche Ausarbeitung Thema Datenkompression

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

eine effiziente und schnelle Übermittlung von Informationen ermöglicht. Dadurch konnte in Krisenmomenten schnell reagiert werden, um so größeren Katastrophen frühzeitig abzuwenden, aber leider auch zu verursachen.

Der Ist-Stand

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

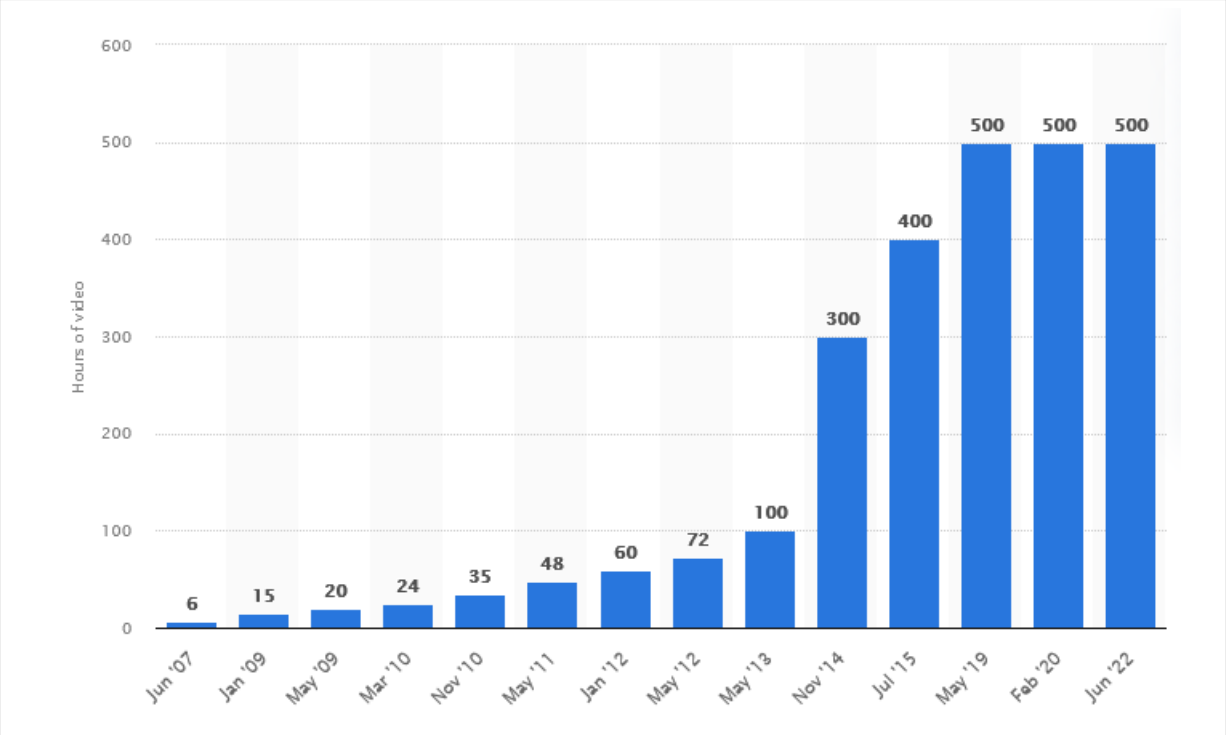

Abb. 1: **Anzahl der Youtube Videos** Die Anzahl an Minuten die auf Youtube hochgeladen werden. Abbildung von Statista:
https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/

zu speichern, benötigt Google riesige Serverfarmen, die auf dem gesamten Globus verstreut sind.

Eine genaue Zahl ist der Öffentlichkeit nicht bekannt, es steht jedoch außer Frage, dass diese nochmal um einiges höher ausfällt, würde es keine Verfahren zur Datenkompression geben.

Ein weiterer Gesichtspunkt ist der eigentliche Nutzen von Youtube, dem Streamen von Videos. Um ein Video sehen zu können, muss dieses von dem Youtube Server, zum Nutzer, dem Client übertragen werden. Durch die Komprimierung der Quelldateien sind die zu übertragenden Daten schon geschrumpft. Es können jedoch noch weitere Schritte absolviert werden, um die Daten für den Nutzer besser zugänglich zu machen. Dazu werden Verfahren wie 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 dekomprimieren und komprimieren verloren geht, sollten die Verfahren verlustbehaftet sein.
In vielen Fällen kann die originale Auflösung vom Endgerät nicht abgespielt werden, und wird deshalb von diesem auf eine niedrigere Auflösung skaliert. Beispielsweise wenn das Endgerät lediglich 1080p auflösen kann, aber ein Video in 4K Auflösung gestreamt werden soll. Trotzdem werden die vollen Daten des Videoformats empfangen. Um diese Verschwendung von Bandbreite zu sparen, wird Transsizing verwendet. Die originalen Daten werden in eine kleinere Auflösung skaliert, und anschließend übertragen.
Um die Bitrate zu minimieren wird Transrating verwendet. So kann die Auflösung beibehalten werden bei jedoch geringere Bitrate. Die Verfahren zur Minimierung des Datenstroms hören sich zunächst sehr mächtig an, sind jedoch mit Vorsicht zu genießen. Der Vorgang ist nämlich Verlustbehaftet und kann bei zu starker Nutzung zu Artefakten führen. Dafür ermöglicht es jedoch Menschen, deren Internetzugang ein Abspielen in hoher Qualität nicht zulässt, den Streaming Anbieter zu nutzen.

Der Ursprung der Datenkompression ist dementsprechend der Weiterentwicklung des Morse Codes zurückzuführen.

TODO

1.2 Steigende Komplexität

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

1.3 Ziel der Arbeit

2 Grundlagen

test . . .

2.1 Brotli Kompressionsstandard

Brotli-G ist eine Weiterentwicklung des Brotli Kompressionsstand, der von AMD im Jahre 2022 entwickelt und veröffentlicht wurde. Die AMD Spezifikation bietet parallele Datenverarbeiten nach dem SIMD Prinzip (Kap. 2) auf Parallelrechnern, wie GPUs und Multithreaded CPUs. Um das von AMD veröffentlichten Kompressionsmodell zu verstehen, muss zunächst das Original untersucht werden.

Brotli ist ein von Google Research entwickelter Kompressionsstandard, der 2013 veröffentlicht wurde. Er ist darauf ausgelegt, Webinhalte effizienter zu komprimieren als ältere Standards wie Gzip oder Deflate. BrotliG wurde mit bedacht auf Kompatibilität mit dem offiziellen Brotli entwickelt. So sollte Brotli auch in der Lage sein, Inhalte, die mit BrotliG komprimiert wurden, zu entschlüsseln. Zu beachten ist, dass dies nur in diese Richtung funktioniert, und somit Brotli das BrotliG Format nicht dekodieren kann [Bro22].

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

2.1.1 LZ77

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

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

1. Eingabestrom: Die zu kodierenden Daten

2. Symbol: Ein willkürlich gewähltes Element des Eingabestroms
3. Datenfenster: Alle Symbole vom Start des Eingabestroms bis zum aktuell betrachteten Symbol
4. Vorschaubuffer: Alle Symbole von dem aktuelle betrachteten Symbol bis zum Ende des Eingabestroms
5. Vorschaufenster: Ein Buffer fester Größe das Symbole vom aktuell betrachteten Symbol bis zum Ende des Buffers enthält

Der Ablauf des Algorithmus besteht aus folgenden Schritten:

Zu Beginn des Algorithmus wird das Datenfenster auf den Start des Eingabestroms gesetzt. Dieses Fenster ist zunächst leer. Das 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 werden das Daten- und Vorschaufenster um die Lauflänge der kodierten Symbole im Eingabestrom verschoben. [**?**]

Der LZ77-Algorithmus ist einfach und effektiv. Seine Weiterentwicklungen nennen sich LZ78 und LZW, die zwar dem LZ77 Algorithmus technisch überlegen sind, aufgrund von Patenten jedoch nicht so eine große Rolle spielen wie die erste Iteration von Lempel und Ziv. In Kombination mit anderen Techniken wie der Huffman Codierung bildet der LZ77-Algorithmus jedoch die Grundlage vieler leistungsstarker Kompressionsstandards, die heute in vielen Applikationen zu finden sind.

2.2 Parallele Datenverarbeitung

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

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

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

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

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

können kombiniert werden.

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

SISD (Single Instruction, Single Data)

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

SIMD (Single Instruction, Multiple Data)

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

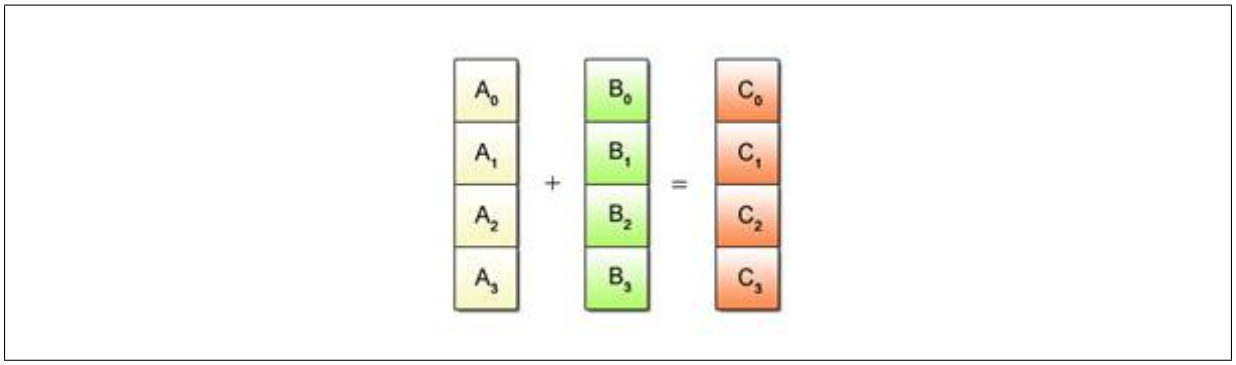

Abb. 2: **Parallele Addition von Daten** In der Abbildung ist eine Addition verschiedener Daten zu sehen. Da die selbe Operation ausgeübt wird können die einzelnen Komponenten parallel verarbeitet werden. Abbildung ist aus http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html

MISD (Multiple Instruction, Single Data)

Um alle Kombinationen von Daten und Instruktionsströmen zu zeigen wurde auch MISD definiert. Die Rechnerarchitektur bezieht sich darauf, das auf nur einem Datenpunkt verschiedene

Operationen ausgeführt werden. Für eine lange Zeit war diese Art von Rechnerarchitektur rein theoretisch anzutreffen, da weder

MIMD (Multiple Instruction, Multiple Data)

Wie auch die SIMD Architektur ist MIMD in Parallelrechnern anzutreffen. Das Operationsprinzip von MIMD ist die Datenparallelität. Das Funktionsprinzip von MIMD-Rechnern umfasst die gleichzeitige Ausführung von Anweisungen durch mehrere Prozessoren, die entweder über gemeinsame Variablen oder durch Nachrichten miteinander kommunizieren [DC96]. TODO später [JBG17]

2.3 Die traditionelle Rendering Pipeline

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

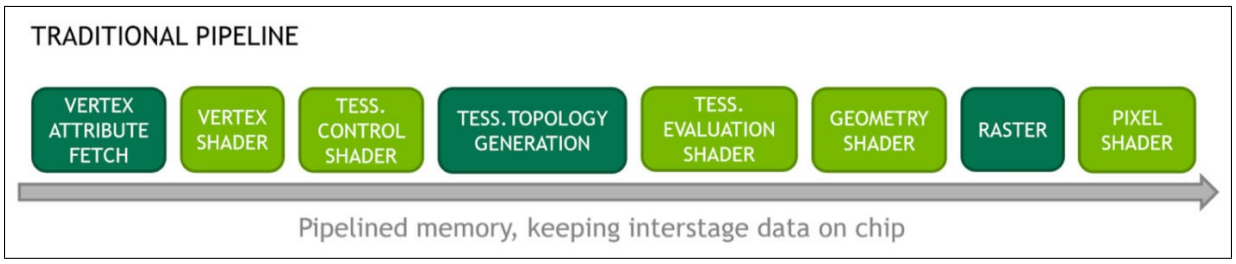

Abb. 3: **traditionelle Rendering Pipeline**
Die Abbildung beschreibt den Verlauf durch die einzelnen Shader Stages, die jeder Vertex macht. Entnommen wurde diese aus dem NVidia Blogpost [Kub18]

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

2.3.1 Vertex Shader

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

2.3.2 Tessellation Stage

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

Tessellation Control Shader (TCS)

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

Tessellation Topology Generation (TPG)

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

Tessellation Evaluation Shader (TES)

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

2.3.3 Geometry Shader

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

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

2.3.4 Pixel Shader

Der Pixel bzw. Fragment Shader ist der letzte programmierbare Schritt der Grafikpipeline. In diesem werden die transformierten Vertices und Primitiven schlussendlich gezeichnet. Der Pixel Shader operiert jedoch nicht auf diesen Daten, sondern auf sogenannten *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 wiederum ein Fragment.

2.4 Compute Shader

Der Compute Shader gehört nicht zur traditionellen Grafikpipeline, kann darin aber seinen Nutzen finden. Sie dienen dazu, jede Mögliche Information die gewünscht ist auf der GPU zu berechnen. Anders als bei den Shader Stages der traditionellen Grafikpipeline (Kap. 3), erwartet der Compute Shader keine definierten Input/Output Daten, wie beispielsweise der Vertex Shader, der als Input und Output einen Vertex erwartet. Der Compute Shader kann also willkürliche Daten verarbeiten und dabei noch die Parallelisierung der GPU nutzen [Com24].

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

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

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

$$numThreadsX, numThreads, numThreadsZ \leq 128$$
$$numThreadsX * numThreadsY * numThreadsZ = 1024$$

(Für Compute Shader Version 5_0)

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

3 Task und Mesh Shader

Die Architektur, auf der die neuartigen RTX-GPUs von Nvidia aufbauen, erweitert die Möglichkeiten, wie die Parallelisierung von GPUs genutzt werden kann. Mit der GeForce RTX 20er Serie wurden die ersten GPUs mit der Turing Architektur veröffentlicht, die sich auch an Privatpersonen richtet. Als großer Verkaufspunkt wurde bereits früh mit den Möglichkeiten von Real-time Raytracing und Deep Learning durch Tensor Core geworben [Bur20]. Eine wesentliche Änderung an der Grafikpipeline wird jedoch bis heute noch relativ wenig Beachtung geschenkt. Mit dem Shader Model 6 hat NVidia ihre sogenannte "next-generation shading Pipeline" vorgestellt. Damit wird eine Alternative zur traditionellen Shading Pipeline gestellt, die dem Entwickler mehr Freiheit überlässt, die Parallelisierbarkeit der GPU zu nutzen. Der Mesh Shader hat die Eigenschaften des Compute Shaders 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 findet dadurch seinen Nutzen auch außerhalb des Renderings [Ile22]. Mit der *Mesh Shading Pipeline* wurde die Möglichkeit der Parallelisierung des Compute Shaders mit der neuen Rendering Pipeline verknüpft. Anders als bei der herkömmlichen 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.

3.1 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 zugeschnitten und in kleine Clustern verarbeitet. Dazu wird der Primitive Distributor vor der Vertex Shader Stage aufgerufen. Dieser liest die Daten des Index Buffers und generiert dementsprechend möglichst performant diese Cluster an Daten. Der Schritt des Primitive Distributor ist jedoch eine fixed-function Stage der Grafikpipeline, wodurch der Entwickler keinen direkten Zugriff hat. Das hat zur Folge, dass die Cluster nicht auf die Bedürfnisse des Entwicklers und dessen Implementierung angepasst werden können. Zuzüglich werden die Cluster zu jedem Frame, bzw. vor jedem Aufruf der Pipeline neu generiert. Dieser Schritt ist redundant, sollte das Dreiecksnetz zur Laufzeit unverändert bleiben [Car22], [Kub18].

Durch die Compute Shading Natur des Mesh Shaders ist der Input der Daten nicht mehr festgelegt wie bei der traditionellen Pipeline. Dadurch kann der Entwickler seine eigenen

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 Auswahl der Meshletgröße ist abhängig von der verwendeten GPU. So wird im NVidia Blogpost „Introduction to Turing Mesh Shaders“ eine maximale Anzahl an Vertices von 64, und Primitiven von 126 empfohlen. Es werden 126 statt 128 Primitiven empfohlen, da 4 Byte für die Anzahl der Primitiven verwendet werden, die im selben Block Speicher enthalten sein sollen, bzw. keinen weiteren Block beanspruchen sollen [Kub18]. Arseny Kapoulkine hat verschiedene Meshletgrößen miteinander verglichen. Er ist zu dem Schluss gekommen, dass 64 Vertices und 84 Primitives am effizientesten ist, insbesondere dann, wenn im Task Shader Culling an den einzelnen Meshlets betrieben wird. Des weiteren ist die Empfehlung des Blogposts nach eigenen Tests zwar ein guter Maßstab, jedoch wird im Durchschnitt viel Speicher des Primitiven Buffers ungenutzt bleiben, da die 126 Primitiven mit 64 Vertices nie erreicht werden [Kap23].

3.2 Implementierung Mesh Shader

Wie im vorherigen Unterkapitel angekündigt muss das Dreiecksnetz auf der CPU zu Meshlets geschnitten werden. Dazu wurde in dieser Arbeit er Meshoptimizer von Zeux verwendet [Zeu]. [Implementierung von Zeux beschreiben] Die Funktion „meshopt_buildMeshlets“ nimmt als Eingabeparameter die maximale Anzahl an Vertices und Primitiven (Kap.3.1), die Vertex und Index Daten sowie drei leere Buffer. Der Buffer *meshlet_indices* wird die neuen Index Daten enthalten, mit denen die Primitiven berechnet werden können. Der *meshlet_vertices* Buffer beinhaltet die einzigartigen Vertices des Dreiecksnetzes (Kap **??**). Der letzte Buffer wird in dieser Arbeit als *Meshlet Descriptor* bezeichnet. Der Einfachheit halber wird er im Code jedoch einfach als *meshlets* implementiert. Der Meshlet Buffer setzt sich auch folgenden Elementen zusammen

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

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

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

3.3 Mesh Shader Implementation

Im Codeabschnitt 1 ist ein einfacher Mesh Shader zu sehen. Im Mesh Shader wird die Root Signature entsprechend den Anforderungen gesetzt. Minimal wird ein StructuredBuffer für jeden der auf der CPU generierten Meshlet Buffer benötigt. Um dem Endresultat 3-Dimensional wirken zu lassen, wird ein ConstantBuffer verwendet, der die*model*, *modelView* und *modelViewProjection* Matrix beinhaltet. Zusätzlich dazu nimmt der Constant Buffer noch ein boolean, um zu steuern, das die Meshlets farbig hervorgehoben werden. 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 mittles *[NumThreads(128, 1, 1)]* im Mesh Shader, oder, falls vorhanden, im Task Shader festgelegt.

```
[RootSignature(ROOT_SIG)]
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void main(
   in uint gtid : SV_GroupThreadID,
   in uint gid : SV_GroupID,
   out vertices VertexOut verts[64],
   out indices uint3 tris[84]
)
{
  Meshlet m = Meshlets[gid];
  uint3 primitive;

  SetMeshOutputCounts(m.VertCount, m.PrimCount);

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

  if (gtid < m.VertCount)
  {
    uint vertexIndex = GetVertexIndex(m, primitive[0]);
    verts[gtid] = GetVertex(gid, vertexIndex);
  }
}
```

Code 1: Mesh Shader Main

3.4 Das auf der GPU zu dekodierende Binärformat

[Eventuell nicht als eigenes Unterkapitel]

Literaturverzeichnis

[Bro22] 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/*

[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)

[Com24] Compute Shader. In: *OpenGL Wiki* ((besucht am 09.02.2024)). *https://www.khronos.org/opengl/wiki/Compute_Shader*

[CR12] COZZI, P. ; RICCIO, C.: *OpenGL Insights*. Taylor & Francis, 2012 (Online access with subscription: Proquest Ebook Central). *https://books.google.de/books?id=CCVenzOGjpcC*. – ISBN 9781439893760

[DC96] In: DAL CIN, Mario: *Klassifizierung von Rechnerarchitekturen*. Wiesbaden : Vieweg+Teubner Verlag, 1996. – ISBN 978–3–322–94769–7, 22–32

[Ile22] In: ILETT, Daniel: *Advanced Shaders*. Berkeley, CA : Apress, 2022. – ISBN 978–1–4842–8652–4, 517–582

[JBG17] JAKOB, Johannes ; BUCHENAU, Christoph ; GUTHE, Michael: A Parallel Approach to Compression and Decompression of Triangle Meshes using the GPU. In: *Computer Graphics Forum* 36 (2017), Nr. 5, 71-80. *http://dx.doi.org/https://doi.org/10.1111/cgf.13246*. – DOI https://doi.org/10.1111/cgf.13246

[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*

[Zeu] ZEUX: meshoptimizer. *https://github.com/zeux/meshoptimizer*

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Titel

_____

_____

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

_____
Ort

_____ _____
Datum Unterschrift