HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg
Fakultät Elektrotechnik und Informatik

Studiengang: Informatik

Bachelorarbeit

# Parallel Mesh Decompression on the GPU

Janek Foote

Abgabe des Arbeit: 04.02.2024

Betreut durch:

Quirin Meyer, Hochschule Coburg

# Inhaltsverzeichnis

# 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 aufs Papier (oder eher auf den Bildschirm) bringen. Selbst im Ikea kann die Couch die einem Gefällt an einem Tablet Editor weiter konfiguriert werden. Künstler wollen Modelle erschaffen, die den Eindruck gewinnen wollen, Realitätsgetreu zu sein. Die Folge davon ist, das diese Modelle stetiger komplexer werden, und somit ein größerer Speicheraufwand benötigt wird. Um dem Entgegenzuwirken gibt es Möglichkeiten, um digitale Signale zu komprimieren. Ursprünglich entwickelt zur Repräsentation von Daten 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 den Signalsequenzen können ganze Sätze übermittelt werden. Das bekannteste Werkzeug für den Morse Code ist der Telegraph, 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 in 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 zu betrachten (beispielsweise vorherige Buchstaben). Da diese Häufigkeiten abhängig vom Alphabet sind, sollten diese nicht übergreifend verwendet werden. So sind die Buchstaben „E“ und „T“ die Buchstaben im Englischen Alphabet, welche die höchste Auftrittswahrscheinlichkeit besitzen. Der Ursprung der Datenkompression ist zu der Weiterentwicklung des Morse Codes zurückzuführen. Morse Code ist eine Form der Huffman Codierung

## 1.1 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.

## 1.2 Kompression

Brotli BrotliG

# 2 Grundlagen

test ...

## 2.1 Compute Shader

Compute Shader sind kein fester Bestandteil der Rendering Pipeline. Sie dienen als ..., um auf der GPU Operationen performant auszuführen, die auf CPU Ebene länger brauchen würde. Compute Shader blablabla Um das SIMD Konzept des Compute Shaders zu verstehen sind zwei Variablen elementar wichtig. SVGroupThreadID und SVGroupID. Um einen aktiven Compute Shader mittels CPU aufzurufen, wird die Dispatch Methode aus der verwendeten Grafik-API verwendet. 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)

## 2.2 Primitive Subgroups

## 2.3 Parallele Datenverarbeitung

Michael Flynn unterteilte Rechnerarchitekturen in vier Kategorien. Abhängig davon wurden Daten un [JBG17] SIMD MIMD

## 2.4 Die traditionelle Rendering Pipeline

Um den Nutzen der neu vorgestellten Task- und Mesh-Shader Pipeline zu verstehen, muss zunächst die traditionelle Pipeline da betrachtet werden, wo sie verbessert werden kann.

### 2.4.1 Vertex Shader

Zunächst wird der vom Entwickler programmierbare Vertex Shader angesteuert. Hier können Operationen auf den einzelnen Vertex ausgeführt werden. Der Vertex Shader wird für jeden Vertex einzeln aufgerufen. Hier zeichnet sich das SIMD Modell der GPU aus (**??**, da die Instruktionen parallel auf mehrere Daten gleichzeitig ausgeführt werden.

# 3 Task und Mesh Shader

Die Architektur auf die 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 2.1, 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, sogennante Meshlets, zu unterteilen. Die traditionelle shading pipeline verarbeitet die Daten des Dreiecksnetzes in serieller Manier. Dadurch kommt es jedoch zu bottlenecks. In der traditionellen Pipeline werden Vertex und Primitiven Daten zugeschnitten und in kleinen Clustern verarbeitet. Dazu wird der Primitive Distrubuter vor der Vertex Shader Stage aufgerufen. Dieser liest die Daten des Index Buffers und generiert dem entsprechend möglichst performant diese Cluster an Daten. Der Schritt des Primitive Distrubuters ist jedoch ein fixed-function step der Grafikpipeline, wodurch der Entwickler keinen direkten Zugriff hat. Das hat zur Folge, das 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 Shader neu generiert. Dieser Schritt ist redundant, sollte das Dreiecksnetz zur Laufzeit unverändert bleiben [Car22], [Kub18].

Durch die Compute Shading 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 Vertexanzahl von 64, und Primitivenanzahl 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, das 64 Vertices und 84 Primitives am effizientesten ist, insbesondere dann, wenn im Task Shader Culling an den einzelnen Meshlets betrieben wird. Desweiteren 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 2.2). 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 **??** ist der in dieser Arbeit verwendetete 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 auf die Szene anzupassen wird zusätzlich noch 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.

```
\label{lst:shadercode}
[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 Speichern als Binärformat

[Eventuell nicht als eigenes Unterkapitel]

# Literaturverzeichnis

[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)

[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