

# Aufbau einer modernen Rendering-Pipeline

Proseminar-Ausarbeitung von

**Jonas Heinle**

An der Fakultät für Informatik  
Institut für Visualisierung und Datenanalyse,  
Lehrstuhl für Computergrafik

24. Mai 2019

# Inhaltsverzeichnis

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Abstract</b>  | <b>1</b>  |
| <b>2</b> | <b>Einleitung</b>  | <b>2</b>  |
| <b>3</b> | <b>Moderne Rendering-Pipeline</b>                            | <b>4</b>  |
| 3.1      | Anwendung . . . . .  | 4         |
| 3.2      | Geometrie Stufe . . . . .                                    | 5         |
| 3.2.1    | Vertex Shader . . . . .                                      | 5         |
| 3.2.2    | Primitive Assembly . . . . .                                 | 5         |
| 3.2.3    | Tessellation . . . . .                                       | 5         |
| 3.2.4    | Geometry Shader . . . . .                                    | 5         |
| 3.2.5    | Clipping . . . . .   | 5         |
| 3.2.6    | Viewport Transform . . . . .                                 | 6         |
| 3.3      | Rasterisierung . . . . .                                     | 6         |
| 3.4      | Fragment Shader . . . . .                                    | 6         |
| 3.5      | Per Fragment Operations . . . . .                            | 6         |
| 3.5.1    | Multisample Fragment Ops . . . . .                           | 6         |
| 3.5.2    | Stencil Test . . . . .                                       | 6         |
| 3.5.3    | Occlusion Query . . . . .                                    | 6         |
| 3.5.4    | Blending . . . . .   | 6         |
| 3.5.5    | Logical Operations . . . . .                                 | 6         |
| 3.6      | Framebuffer und Buffer Objekte . . . . .                     | 6         |
| 3.7      | GPU Memory . . . . .   | 6         |
| 3.8      | Compute Shader . . . . .                                     | 6         |
| <b>4</b> | <b>Unterschied moderne und klassische Rendering-Pipeline</b> | <b>8</b>  |
| 4.1      | Freie Programmierung oder reines Konfigurieren . . . . .     | 8         |
| 4.2      | Vertex Arrays, Index Buffers, .. . . .                       | 8         |
| <b>5</b> | <b>Ausblick</b>  | <b>9</b>  |
| 5.1      | Raytracing Unterstützung . . . . .                           | 9         |
| 5.2      | Task-/Mesh Shaders . . . . .                                 | 10        |
|          | <b>Literaturverzeichnis</b>                                  | <b>11</b> |

# 1. Abstract

In unserer heutigen hoch technologisierten Welt steigt stetig und rasant die Leistungsfähigkeit moderner Grafikhardware. Um heutzutage Grafik auf einem modernen Computer darzustellen sind mittlerweile eine Vielzahl von Zwischenschritten nötig. Diese Arbeit beschäftigt sich um die einzelnen Stufen dieser Kette, deren jeweilige Aufgabe, ihren Datentransfer untereinander, ihre Reihenfolge sowie ihren Arbeitskontext. Beim Arbeitskontext werden unter anderem die unterschiedlichen Koordinatensysteme untersucht. Dabei soll nicht nur konkret auf die Arbeitsweise der einzelnen Stufe eingegangen werden, sondern auch im Speziellen auf die Zusammenarbeit und Kommunikation. Exemplarische Fragestellungen die behandelt werden sind folgende: Können in dieser Art der Abarbeitung Flaschenhälse entstehen und wie werden Sie umgangen bzw. bekämpft. Welchen Einfluss hat der Programmierer auf die Pipeline bzw. welche Schritte kann er selber implementieren und welche Schritte werden rein von der Hardware übernommen und können nicht von ihm modifiziert werden. Diese Arbeit macht den Aufbau einer modernen Rendering-Pipeline verständlich und vermittelt den Begriff Rasterisierung. Zusätzlich wird es bei dieser Abhandlung, dank des stetigen technologischen Fortschritts, ein Ausblick auf zukünftige Entwicklungen und Neuerungen gegeben, die zukünftig moderne Rendering-Pipelines beherrschen wird.

## 2. Einleitung

Wollen wir mit einer Anwendung 3.1 eine Szene von Objekten, z.B. eine Teekanne in einem Stadion, auf einem Computer darstellen, so liegt das Objekt zuerst in Form von vielen Eckpunkten(Dreiecken, Linien) in der Anwendung vor. Dabei befinden wir uns im "Model Space". Wollen wir die Teekanne innerhalb unserer Welt (das Stadion) bewegen, wenden wir auf jede Koordinate des Modells(gemeint sind alle mit dem Modell assoziierten Eckpunkte, Normalen) den "Model Transform" an und gehen somit in den "World Space". Die geschieht meist in der Anwendung bevor die Geometrie zur Geometriestufe 3.2 geschickt wird.

Als Anwender können wir nun zu Beginn der Geometriestufe mit einem eigenen Vertex Shader die Beleuchtungsberechnung etc. der Eckpunkte bestimmen. Dabei helfen die von der Anwendung mitgegebenen Attribute. Auch in welchen Koordinatensystem dies geschieht bleibt dem Programmierer überlassen. In der anschließenden Stufe werden diese Eckpunkte zu Primitiven zusammengesetzt und können weiter unterteilt werden 3.2.3. Die Primitive werden von dem Tessellation- zum Geometry Shader durchgereicht, welcher diese vervielfachen, entfernen oder beliebig verändern kann..

Um eine (orthographische oder auch perspektivische) Projektion vorzubereiten, wenden wir einen "View Transform" an. Damit können wir in den "View Space" gelangen und vereinfachen die darauffolgende Projektion. Egal welche Art von Projektion wir angewendet haben, unsere Szene mit der Teekanne im Stadion liegt nun in einem Einheitswürfel.

Nun kann die Teekanne derart platziert sein, dass Teile außerhalb unseres Sichtfeldes(frustum) liegen. Aufgabe des Clipping 3.2.5 ist nun das "Frustum Culling", also das Abschneiden des nicht sichtbaren Bereichs, um weitere Berechnungen innerhalb darauffolgender Stufen zu beschleunigen. Der Entwickler hat keinen Einfluss auf das Clipping 3.2.5 (Abschneiden nicht relevanter Szenengeometrie). Ein Algorithmus der dieses Problem löst, ist der Cohen-Sutherland-Algorithmus.

Nach dem Beschneiden liegen nur noch die sichtbaren "Primitive" vor und gelangen in die nächste Pipeline-Stufe, dem Viewport Transform 3.2.6. Die x- und y-Komponente unserer Eckpunktpositionen werden auf die Bildschirmgröße skaliert und bilden die schlussendliche Position des Vertex. Dabei ist es wichtig anzumerken, dass wir die z-Komponente für weitere Berechnungen in der Rasterisierung 3.3 speichern.

Nun sind wir an dem Punkt angelangt, an dem wir unsere transformierten, projizierten, sichtbaren Eckpunkte mit ihren Shadinginformationen vorliegen haben (Außerdem auch

Tiefenwert!). Ziel der Rasterisierung ist es nun jeden einzelnen Bildschirmpixel mit diesen Informationen einzufärben. Vor dem endgültigen Darstellen der Bildschirmpixel lassen sich auf den zuvor erzeugten Fragmenten eine Vielzahl von Operationen 3.5 ausführen. So haben wir nun das finale Bild produziert.

Besonders interessant sind die neuen Shader für Ray Tracing. Die neuen Einheiten bilden eine Ergänzung der bisherigen Rasterisierung 5.1.

## 3. Moderne Rendering-Pipeline

### 3.1 Anwendung

Zu rendernde Objekte werden zunächst von der Anwendung zur Grafikkarte geschickt. Dabei liegen die Daten über ein Objekt beispielhaft im OBJ-Dateiformat vor. Die Informationen über einen Vertex, das sind die Position, Normale und Texturkoordinaten sind für Berechnungen der nächsten Stufe, des Vertex Shaders 3.2.1, wichtig. Desweiteren werden Kamera- und Lichtposition in die Geometriestufe 3.2 übergeben. In der Anwendung können Informationen über ein Objekt geupdatet werden, z.B. bei Kollisionen von Objekten bei denen sich Positionen verändern. Allgemein werden hier Benutzereingaben gesammelt und verarbeitet. Als programmierbare Stufe dieser Pipeline kann der Anwender konkrete Verbesserungen in diese Stufe einbringen um z.B. mit einem BVH die Anzahl der Primitive für die Geometriestufe zu verringern.

```
4 newmtl Material
5 Ns 96.078431
6 Ka 1.000000 1.000000 1.000000
7 Kd 0.640000 0.640000 0.640000
8 Ks 0.500000 0.500000 0.500000
9 Ke 0.000000 0.000000 0.000000
10 Ni 1.000000
11 d 1.000000
12 illum 2
13
```

Abbildung 3.1: Definition beispielhaftes Material

```
1 # Blender v2.79 (sub 0) OBJ File: ''
2 # www.blender.org
3 mtlib Suzanne.mtl
4 o Suzanne
5 v 0.437500 0.164062 0.765625
6 v -0.437500 0.164062 0.765625
7 v 0.500000 0.093750 0.687500
8 v -0.500000 0.093750 0.687500
9 v 0.546875 0.054688 0.578125
```

Abbildung 3.2: Vertex Daten im OBJ-Dateiformat

## 3.2 Geometrie Stufe

### 3.2.1 Vertex Shader



Abbildung 3.3: Funktionsweise Vertex Shader

Hier arbeiten wir komplett in Objektkoordinaten.

### 3.2.2 Primitive Assembly

### 3.2.3 Tessellation

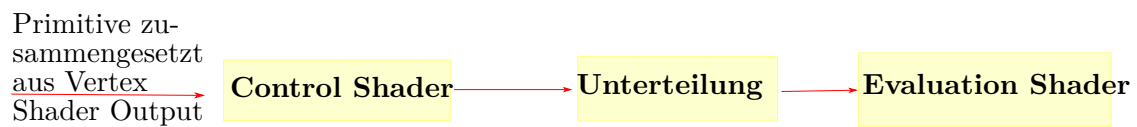


Abbildung 3.4: Funktionsweise Tessellation Shader

Hinzufügen neuer Vertices.

### 3.2.4 Geometry Shader



Abbildung 3.5: Funktionsweise Geometry Shader

### 3.2.5 Clipping

Um Objekte bzw. Objektausschnitte, welche außerhalb des Sichtfensters liegen, für die Bildsynthese zu verwerfen kommt nun das Abschneiden(englisch = "clipping"). Algorithmus von Sutherland-Hodgman. Wir gehen vom Clip-Space zum Window-Space

### 3.2.6 Viewport Transform

## 3.3 Rasterisierung



Abbildung 3.6: Ablauf Rasterisierung

In der vorherigen Geometriestufe wurde einem klar, dass wir ein Objekt nach dem Anderen betrachten (Object-ordered Rendering)

### 3.4 Fragment Shader

### 3.5 Per Fragment Operations

#### 3.5.1 Multisample Fragment Ops

#### 3.5.2 Stencil Test

#### 3.5.3 Occlusion Query

#### 3.5.4 Blending

#### 3.5.5 Logical Operations

### 3.6 Framebuffer und Buffer Objekte

### 3.7 GPU Memory

### 3.8 Compute Shader

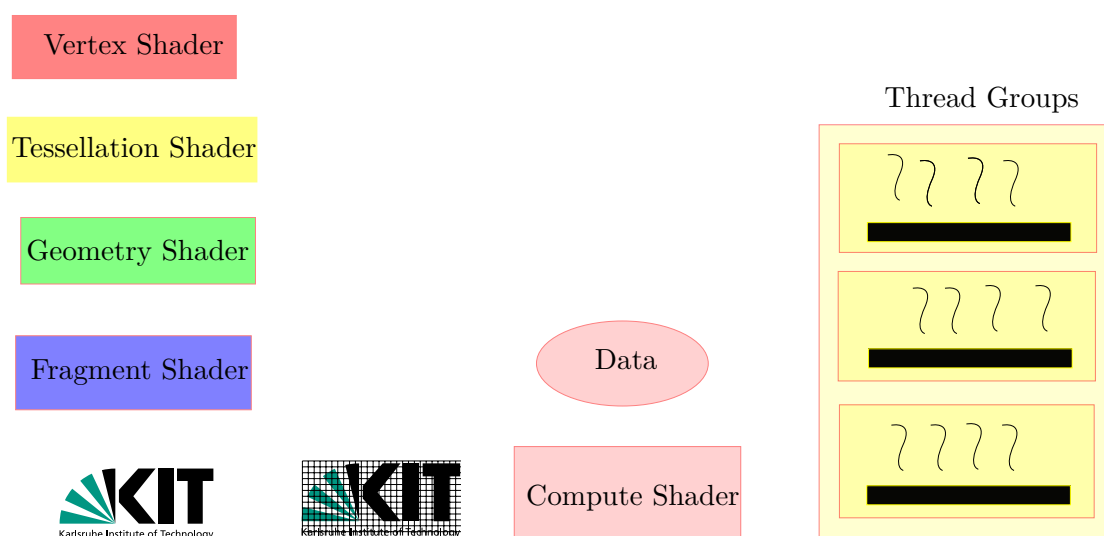


Abbildung 3.7: Funktionsweise Compute Shader



Mit der rasant steigenden Leistung heutiger Grafikkhardware stieg auch der Wunsch beim Anwender nach mehr Rechenleistung. Parallelsierbare Arbeit wird in Threadgroups eingeteilt. Threads innerhalb einer Gruppe laufen gleichzeitig, wohingegen die Threadgruppen untereinander dies nicht müssen.

## 4. Unterschied moderne und klassische Rendering-Pipeline

### 4.1 Freie Programmierung oder reines Konfigurieren

Abbildung 4.1: OpenGL  $\leq$  2.0

Abbildung 4.2: Modernes OpenGL

### 4.2 Vertex Arrays, Index Buffers, ..

## 5. Ausblick

### 5.1 Raytracing Unterstützung

In heutigen modernen Grafikprogrammierschnittstellen(Vulkan, DirectX) befindet sich Raytracing-Funktionalität. Raytracing Ansätze gehen von der objektbasierten (siehe Rasterisierung) zu der Image-ordered Bilderstellung über. Hiermit werden nicht nur Primärsondern auch Sekundärstrahlen(etc.) betrachtet und somit unter Anderem Schatten und Spiegelungen. Wir können diese neuen Shader wie Compute Shader 3.8 programmieren.

| Shader         | Beschreibung   |
|----------------|--|
| Any Hit        | Kann ausgeführt werden wenn wir einen Schnitt mit unserem Strahl gefunden haben. Soll bei transparenten Objekten dazu führen, früher abbrechen zu können.  |
| Callable       | Kann von einem anderen Shader aufgerufen werden.   |
| Closest Hit    | Wird ausgeführt wenn ein Standardstrahl von einem Ray Generation Shader verschossen wurde, und ein "nächster Schnittpunkt" gefunden wurde. Hier kann der Programmierer die Methode <i>shade()</i> implementieren. Dabei sollte auf Schatten getestet, weitere Reflektions- und Transmission miteinander berechnet werden |
| Intersection   | Wird aufgerufen, falls wir eine Bounding Box unserer Beschleunigungsstruktur treffen.  |
| Miss           | Wird aufgerufen, falls keine Szenengeometrie getroffen wurde.  |
| Ray Generation | Mit <i>TraceRay()</i> können wir Strahlen verschießen. Normalerweise versendet man einen Strahl pro Pixel. Wir können jedwedigen Typ von Strahl damit verschießen(Schatten, Primär, Transmit)  |

Tabelle 5.1: Shadereinheiten nach DirectX12.

Hierbei führt ein verschossener Strahl eine Datenstruktur("*payload*") mit sich, die eine

variable Menge an Informationen speichern kann. Als wichtigste zu nennen ist die Distanz bei einem Hit.

Aktuelle Bemühungen gehen nun daran Raytracing und Rasterisierung zu kombinieren. Barré-Brisebois [BBHW<sup>+</sup>19] stellte mit dem Spiel *PICA PICA* eine solche Rendering-Pipeline vor, welche mithilfe von Path Tracing arbeitet. Dabei wird der G-Buffer (Texturen die Position, Normalen, Belichtung eines Frames speichern) noch über Rasterisierung berechnet. Direkten Schatten kann man rastern oder raytracen. Diese Option verspricht eine Anpassungsfähigkeit der Pipeline nach Leistungsfähigkeit. Ähnlich können nun Reflexionen, Global Illumination, Ambient Occlusion und Transmission geraytraced oder auf Compute Shader ausgeführt werden. (Wieder je nach Hardwareleistung). Einzig direkte Beleuchtung sowie Post-Processing Effekte laufen nur über Compute-Shader.

## 5.2 Task-/Mesh Shaders

Allgemein lässt sich an dieser neuen Technik der Trend von fixen Vorgängen innerhalb der Pipeline zu mehr flexibleren wiedererkennen. So kann der Task Shader mit der Control Shader Einheit der bisherigen Tessellation verglichen werden. Jedoch arbeiten wir hier mit mehreren Threads und mit frei wählbaren In- und Output! Mehr Flexibilität ist die Devise. Anstatt einzelne Dreiecke einzeln zu berechnen wie in der bisherigen Pipeline können wir mehrere gleichzeitig bearbeiten in sogenannten parallelen Thread-Gruppen. Hier wird also nichts anderes als das Modell von Compute Shader genommen und auf die Grafikpipeline angewandt. Vorallem bei komplexen Szenen mit vielen Millionen Dreiecken verspricht man sich eine Entlastung der CPU. Durch diesen Aufbau können wir bereits im Vorfeld viele Dreiecke vom Rendern ausschließen.

Dabei wird ein komplexes Mesh in einzelne "Meshlets" zerlegt, welche wiederum von Mesh Shadern behandelt werden.

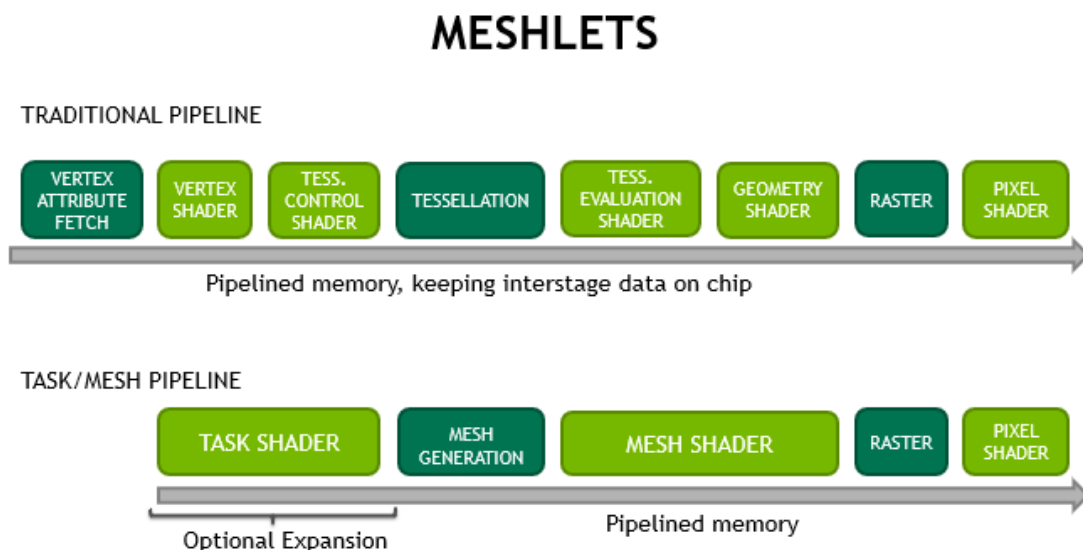


Abbildung 5.1: Neue Form des Task-/Meshshaders[tas19]

# Literaturverzeichnis

- [ABB18] Johan Andersson und Colin Barré-Brisebois: *Shiny Pixels and Beyond: Real-Time Raytracing at SEED*. In: *NVIDIA Sponsored Session, Game Developers Conference*, 2018.
- [AMHH18] Tomas Akenine-Moller, Eric Haines und Naty Hoffman: *Real-time rendering*. AK Peters/CRC Press, 2018.
- [BBHW<sup>+</sup>19] Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak und Johan Andersson: *Hybrid Rendering for Real-Time Ray Tracing*, Seiten 437–473. Apress, Berkeley, CA, 2019, ISBN 978-1-4842-4427-2. [https://doi.org/10.1007/978-1-4842-4427-2\\_25](https://doi.org/10.1007/978-1-4842-4427-2_25).
- [Boy08] Chas Boyd: *The DirectX 11 compute shader*. ACM SIGGRAPH. Cité page, 25, 2008.
- [Dir19] *DirectX12API*. Internet, May 2019. <https://docs.microsoft.com/en-us/windows/desktop/direct3d12/direct3d-12-raytracing>.
- [F<sup>+</sup>04] Randima Fernando *et al.*: *GPU gems: programming techniques, tips, and tricks for real-time graphics*, Band 590. Addison-Wesley Reading, 2004.
- [KWM16] David B Kirk und W Hwu Wen-Mei: *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
- [MS15] Steve Marschner und Peter Shirley: *Fundamentals of computer graphics*. CRC Press, 2015.
- [Ni09] Tianyun Ni: *Direct Compute: Bring GPU computing to the mainstream*. In: *GPU Technology Conference*, Seite 23, 2009. <http://on-demand.gputechconf.com/gtc/2009/presentations/1015-Features-Advantages-DirectCompute.pdf>.
- [tas19] *Mesh Shader*. Internet, May 2019. <https://devblogs.nvidia.com/introduction-turing-mesh-shaders/>.



# Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den 24. Mai 2019

(Jonas Heinle)