

Aufbau einer modernen Rendering-Pipeline

Proseminar-Ausarbeitung von

Jonas Heinle

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

21. Mai 2019

Inhaltsverzeichnis

1 Abstract	1
2 Einleitung	2
3 Moderne Rendering-Pipeline	3
3.1 Anwendung	3
3.2 Geometrie	3
3.2.1 Vertex Shader	3
3.2.2 Primitive Assembly	3
3.2.3 Tessellation	3
3.2.4 Geometry Shader	3
3.2.5 Clipping	3
3.2.6 Viewport Transform	4
3.3 Rasterization	4
3.4 Fragment Shader	4
3.5 Per Fragment Operations	4
3.5.1 Multisample Fragment Ops	4
3.5.2 Stencil Test	4
3.5.3 Occlusion Query	4
3.5.4 Blending	4
3.5.5 Logical Operations	4
3.6 Framebuffer und Buffer Objekte	4
3.7 GPU Memory	4
3.8 Compute Shader	4
4 Unterschied moderne und klassische Rendering-Pipeline	6
4.1 Freie Programmierung oder reines Konfigurieren	6
4.2 Vertex Arrays, Index Buffers,	6
5 Ausblick	7
5.1 Raytracing Unterstützung	7
5.2 Task-/Mesh Shaders	7
Literaturverzeichnis	9

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 ein Objekt, z.B. eine Teekanne, auf einem Computer darstellen, so liegt das Objekt zuerst in Form von vielen Eckpunkten(Dreiecken) in der Anwendung vor. Die Eckpunkte gehen zuerst durch den Vertex-Shader und durchlaufen dort Transformationen. Sind diese aus der Vertex Shader - Einheit werden Sie zu Primitiven(meist Dreiecken) zusammengefasst. Der anschließende Tessellation-Shader nimmt die Primitive entgegen und kann Sie weiterhin unterteilen. Die Primitive werden von dem Tessellation- zum Geometry Shader durchgereicht, welcher diese vervielfachen, entfernen oder beliebig verändern kann. Darauf folgt das Clipping, worauf der Entwickler keinen Einfluss hat und nicht relevante Szenengeometrie entfernt. Nun kommt die Rasterisierung zum Einsatz, welche unsere 3D-Objekte auf den 2D-Schirm bringt und uns Fragmente(Bildschirmpixel) liefert. Auf diesen Fragmenten lassen sich eine Vielzahl von Fragmentoperationen ausführen. Die heutige moderne Rendering-Pipeline zeichnet sich durch seine gesteigerte Flexibilität gegenüber der Älteren, welche viele feste Funktionen beinhaltete, aus. Und dieser Trend scheint nicht abzubreßen. Auch die heutigen rasanten technologischen Fortschritte, z.B. bei der Hardware, erlauben den Einsatz von Technologien, welche früher nicht eingesetzt werden konnten, und geben dem Entwickler immer mehr Freiheiten mit deren Benutzung. So feiert derzeit das Raytracing innerhalb der Echtzeitcomputergrafik einen Siegeszug und wackelt am Thron der bisherigen sehr effizienten Rasterisierung.

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 Texturkoordinate sind für Berechnungen der nächsten Stufe, des Vertex Shaders, wichtig. Des Weiteren sind Flächeninformationen, die für das Primitive-Assembly von Bedeutung sind, und ein Material definiert. In der Anwendung können Informationen über ein Objekt geupdatet werden, z.B. bei Kollisionen von Objekten bei denen sich Positionen verändern,

```
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 mtllib 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

3.2.1 Vertex Shader

Hier arbeiten wir komplett in Objektkoordinaten.

3.2.2 Primitive Assembly

3.2.3 Tessellation

Hinzufügen neuer Vertices.

3.2.4 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

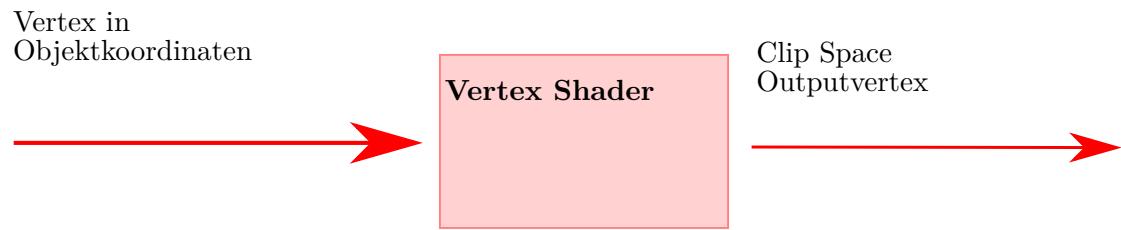


Abbildung 3.3: Funktionsweise Vertex Shader

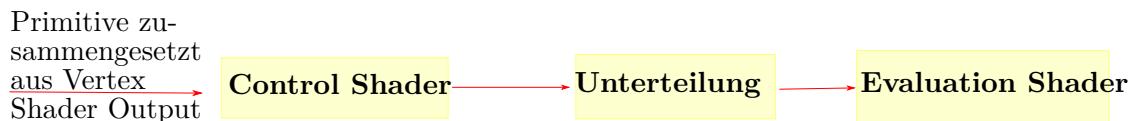


Abbildung 3.4: Funktionsweise Tessellation Shader

3.2.6 Viewport Transform

3.3 Rasterization

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

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

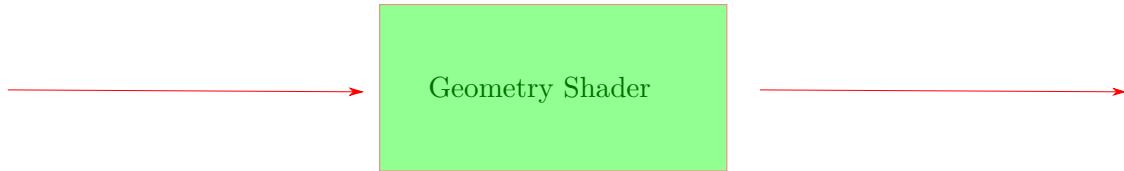


Abbildung 3.5: Funktionsweise Geometry Shader

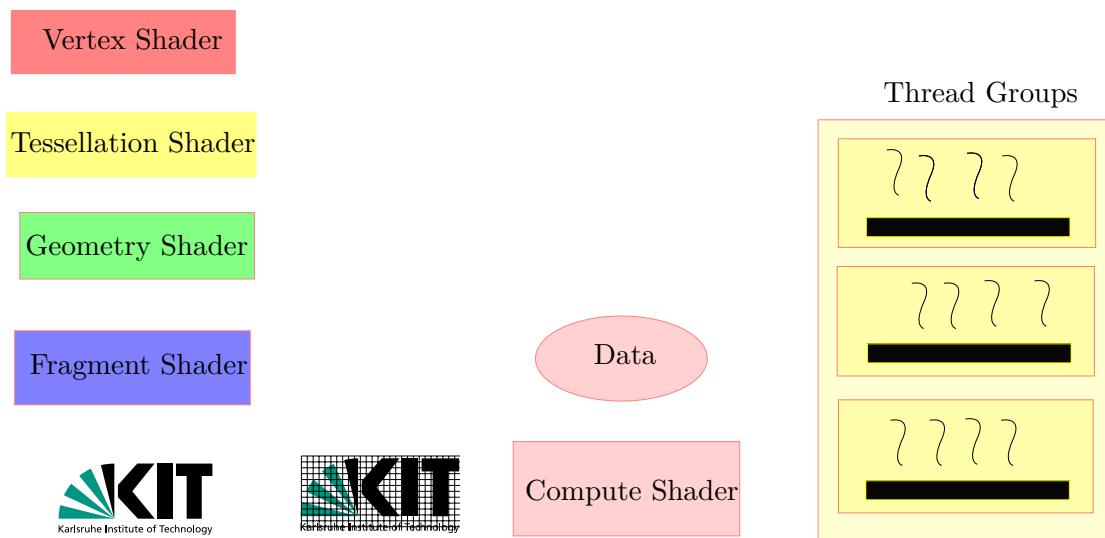


Abbildung 3.6: Funktionsweise Compute Shader

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

Moderne Ansätze gehen von der objektbasierten (siehe Rasterisierung) zu der Image-ordered Bilderstellung über. Hiermit werden nicht nur Primär- sondern auch Sekundärstrahlen(etc.) betrachtet und somit unter Anderem Schatten und Spiegelungen erreicht ohne dafür spezielle Techniken verwenden zu müssen (siehe Rasterisierung).

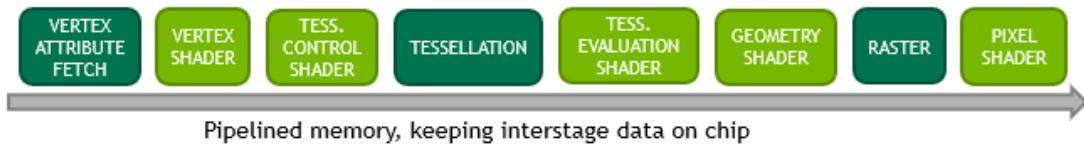
5.2 Task-/Mesh Shaders

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. Vor allem 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.

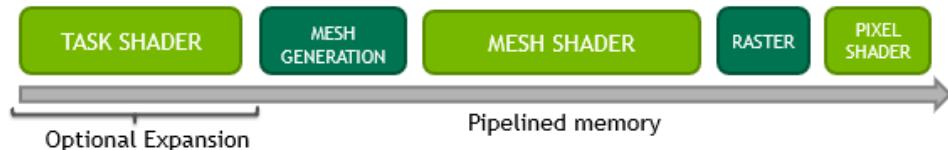
Dabei wird ein komplexes Mesh in einzelne "Meshlets" überlegt, welche wiederum von Mesh Shadern behandelt werden.

MESHLETS

TRADITIONAL PIPELINE



TASK/MESH PIPELINE



Literaturverzeichnis

- [AMHH18] Tomas Akenine-Moller, Eric Haines und Naty Hoffman: *Real-time rendering*. AK Peters/CRC Press, 2018.
- [Boy08] Chas Boyd: *The DirectX 11 compute shader*. ACM SIGGRAPH. Cité page, 25, 2008.
- [F⁺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>.

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 21. Mai 2019

(Jonas Heinle)