Karlsruhe Institute of Technology

Aufbau einer modernen Rendering-Pipeline

Proseminar-Ausarbeitung von

Jonas Heinle

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

28. Mai 2019

Inhaltsverzeichnis

1. Abstract

In unser 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 Transformän 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 Transformän. 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 3.2.6 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 3.2.6 (Abschneiden nicht relevanter Szenengeometrie). Ein Algorithmus der dieses Problem löst, ist der Cohen-Sutherland-Algorithmus.

Nach dem Beschneiden liegen nur noch die ßichtbaren"Primitive vor und gelangen in die nächste Pipelinestufe, dem Viewport Transform 3.2.7. 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 3.4 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 Texturkoordinate sind für Berechnungen der nächsten Stufe, des Vertex Shaders 3.2.2, 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   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 Stufe

3.2.1 Primitive Assembly

Aufgabe ist das Zusammensetzen der Eckpunkte, welche in einem Strom aus dem Vertex Shader kommen. Auf diesen Primitiven(*Patches*, also Punkte, Linien, Drei- oder Vierecke) arbeiten die darauffolgenden Tessellation und Geometry Shader. So sind wir damit beispielsweiße in der Lage in nur einem Draw Call ein Objekt, mit möglicherweise leicht variierenden Attributen, mehrmals zu zeichnen(Instancing)

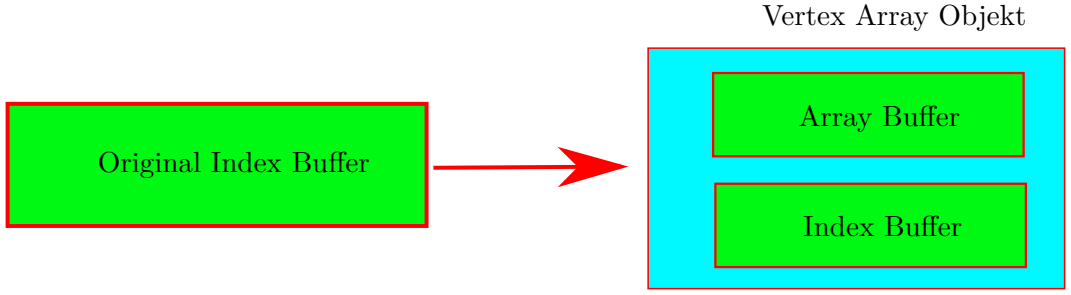

Abbildung 3.3: Buffer Objekte

3.2.2 Vertex Shader

Abbildung 3.4: Ablauf Vertex Shader

Interessant zeigt sich, was direkt vor dem Aufruf des Vertex Shaders passiert: input/primitive assembling. Der Shader behandelt/verändert oder ignoriert als freiprogrammierbare Stufe der Geometry Stage nur die ankommenden Vertices einzeln mit ihren Attributen (jedoch keine Erzeugung oder Löschung!!) mit Hilfe von Daten (Lichtposition, Transformationsmatrix). Dabei kann kein Eckpunkt auf die Informationen eines Anderen zugreifen. Da erlaubt auch die massive parallele Ausführung auf vielen Punkten gleichzeitig. Mit einem Eckpunkt kommt seine Normale, Position, Farbe, möglicherweise Texturkoordinate. Ein minimalistischer Vertex Shader gibt seine Position aus. Dabei wechseln wir vom Modelkoordinaten zu homogenen Clipkoordinaten. Dabei kann ein Vertex Shader mehr als nur die neu berechnete Position zurückgeben. Mit kreativen Ansätzen lassen sich z.B. prozedurale Bewegungen nachhahmen[Eng02]. Es können auch Teile der Leuchtungsberechnung ("Gouraud shading") hier angewandt werden. Das Ergebnis kann zum Tessellation oder Geometry Shader geschickt werden, abgespeichert oder direkt rasterisiert werden.

3.2.3 Tessellation

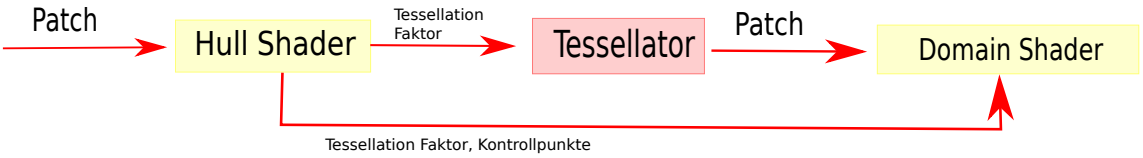

Abbildung 3.5: Funktionsweise Tessellation Shader

Ist eine optionale teil-programmierbare Stufe zur Geometrievervielfachung, d.h. eingehendes Primitiv wird weiter in viele Dreiecke unterteilt. Der *Hull Shader* ist voll programmierbar und damit ist die Anzahl der entstehenden Dreiecke direkt beeinflussbar. Dafür reicht er den Tessellation Faktor zum *Fixed Function Tessellator*. Außerdem lassen sich Kontrollpunkte hinzufügen/entfernen. Dabei lassen sich zwei jeweils benachbarte Patches kontrolliert unterteilen durch geteilte Kontrollpunkte. Dadurch ist sichergestellt, dass keine Artefakte entstehen, wodurch der Übergang der Patches verschattet werden. Der *Tessellator* nimmt die Patches entgegen und unterteilt Sie auf die Art und Weise wie vom *Hull Shader* vorgegeben. Mit dem *Domain Shader* kann man jeden Vertex der vom Unterteiler kommenden Patches nachbearbeiten. Bevor dieser nun die Patches zur nächsten Stufe, dem Geometry Shader (optional) oder dem Fragment Shader schickt, berechnet er die Primitivattribute, Normale, Farbe, Texturkoordinate. Diese Einheit bietet den Vorteil Arbeitslast von der langsamen CPU zur schnelleren GPU zu leiten, indem man niedriger aufgelöste Meshes benutzt und diese dann auf der GPU im Tessallator höher auflöst.(oder

auch Meshes, welche sind schnell deformieren/bewegen). Diese Eigenschaft der Unterteilung ermöglicht auch Verfahren wie Level of Detail oder Anpassungsfähigkeiten an die Leistungsfähigkeit der GPU (viele Dreiecke bei hoher Leistungskapazität).

3.2.4 Geometry Shader

Mit dem Geometry Shader lässt sich jedes Primitiv in ein beliebiges anderes Primitiv verwandeln. Dabei ist seine gedachte Funktion das Verändern der einkommenden Daten(Position, Farbe,..) sowie das Kopieren. Das *instancing* kann auch hier auf einem beliebigen Objekt mehrfach angewandgt werden.

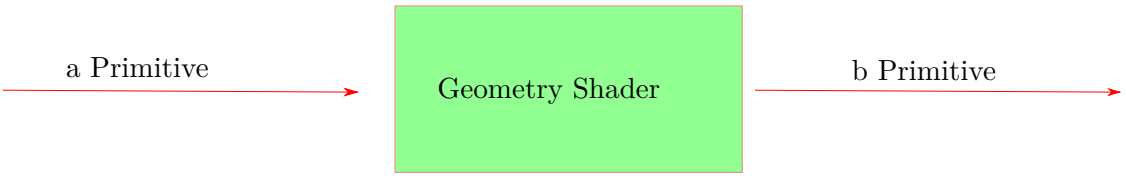

Abbildung 3.6: Funktionsweise Geometry Shader

Ist wiederrum eine optionale frei-programmierbare Einheit die auf den zusammengesetzten Primitiven arbeitet. Eine Vervielfachung ist theoretisch möglich, sollte aber als Aufgabe im Tessellation Shader erbracht werden. Vielmehr können wir diesen Shader einsetzen, um eine Geometrie gleichzeitig auf mehrerer rendertargets zu rendern oder wir machen transform Feedback: Buffer Objects dienen zur Speicherung von Informationen von eingehenden Primitiven. Diese können in weiteren Durchgängen wieder benutzt werden. In der Praxis hat sich der *Geometry Shader* als zu starr, ineffektiv erwiesen. Die Weiterentwicklung Mesh Shader 5.2, welche die Leistung der GPU nun besser nutzen soll, wird in einem späteren Kapitel eingegangen.

3.2.5 Projektionstransformation

Die konkrete Projektion kommt erst beim Weglassen der z-Komponente. Als Vorbereitung transformieren wir unsere Eckpunkte und damit unsere Primitive in einen Einheitswürfel((-1,-1,-1),(1,1,1) als Eckpunkte). Konkrete Implementierungen beziehen sich auf DirectX. Nehmen wir unser Sichtfeld als eine Box war, so ergibt sich eine Orthographische Transformation.

$$M = \begin{bmatrix} 2/(r-l) & 0 & 0 & -(r+l)/(r-l) \\ 0 & 2/(t-b) & 0 & -(t+b)/(t-b) \\ 0 & 0 & 1/(f-n) & -n/(f-n) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Abbildung 3.7: Orthographische Projektion

Bei der orthographischen Projektion haben wir eine achsenorientierte, die Szenengeometrie umschließende Box (AABB). Aufgespannt durch Punkt (l,b,n) und (r,t,f).

Nehmen wir unser Sichtfeld als eine beschnittene Pyramide wahr, welche durch eine near und far plane durchzogen wird, so haben wir eine perspektivische Projektion.

$$M = \begin{bmatrix} 2n/(r-l) & 0 & -(r+l)/(r-l) & 0 \\ 0 & 2n/(t-b) & -(t+b)/(t-b) & 0 \\ 0 & 0 & f/(f-n) & -fn/(f-n) \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Abbildung 3.8: Perspektivische Projektion

Near plane wird durch beide Punkte (l,b,n) und (r,t,n) aufgespannt und f ist die z-Koordinate der far plane. Beachte bei der perspektivischen Projektion, dass der Z-Buffer bei größeren Distanzen ungenauer wird(kein lineares Verhalten!). So kann bei einer Szene mit zwei sehr nahliegenden Ebenen erst bei größerer Entfernung Z-Fighting auftreten.

3.2.6 Clipping

Um Objekte bzw. Objektausschnitte, welche außerhalb des Sichtfensters liegen, für die Bildsynthese zu verwerfen kommt nun das Abschneiden. Ein Algorithmus der dies leistet ist von Sutherland-Hodgman. Objekte komplett außerhalb unseres Sichtfensters(abgeschnittene Pyramide bei perspektivischer Projektion) können komplett verworfen werden. Bei Objekten mit Anteilen außerhalb des Fensters liegt es nun daran neue Kontrollpunkte zu setzen umso den Anteil Außerhalb für die Bildsynthese zu verwerfen. Bei perspektivisch korrekter Interpolation ist hier die richtige Reihenfolge zu beachten: Projektionstransformation, Clipping, View-Port-Transform3.2.7!

Außerdem gehen wir vom Clip-Space zum Window-Space.

3.2.7 Viewport Transform

Der auf die sichtbare Szenengeometrie reduzierte Einheitswürfel wird im diesen Schritt auf die Fenstergröße skaliert und zur Rasterisierung weitergegeben.

3.3 Rasterisierung

Zu Beginn dieser Stufe haben wir nun die Eckpunkte der bereits verarbeiteten, transformierten, projizierten Geometrie mit möglichen Beleuchtungsinformationen aus den vorherigen Stufen vorliegen. Zu den Window Coordinates wurde auch der Tiefenwert gespeichert. Mit der Rasterisierung wird nun die Farbe jedes einzelnen Pixels bestimmt. Es ist also die Aufgabe dieser Pipelinestufe herauszufinden welche Geometrie welchen Pixel zu welchen Anteil bedeckt und wie die Shading Informationen zur Farbgebung des Pixels beitragen.

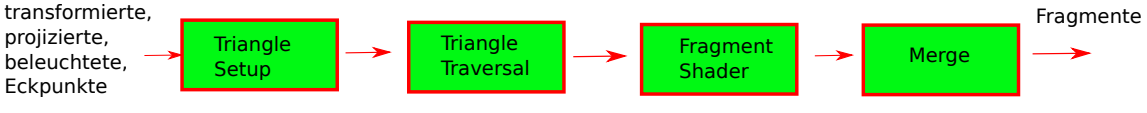

Abbildung 3.9: Ablauf der Rasterisierung

Zuallererst befindet sich die Geometrie im *Triangle Setup*. Unbeeinflussbar vom Programmierer werden hier Daten berechnet, welche zur Pixeleinfärbung benötigt werden. So werden viele zuvor per Vertex berechnete Werte interpoliert(Beleuchtung, Tiefe). Beim darauffolgenden *Triangle Traversal* werden die wichtigen Fragmente erzeugt. Dieser Schritt bestimmt diejenigen Pixel, welche innerhalb des Dreiecks liegen und erzeugt darauf hin die Fragmente für dieses Dreieck anhand der zuvor berechneten/interpolierten per Dreieck Informationen. Als freiprogrammierbare Shadereinheit können im *Pixel Shader* vom Programmierer weitere Berechnungen vorgenommen werden. Dazu zählt eine pro Pixel Beleuchtungsberechnung. Will man Texturen verwenden, so passiert das nun hier.

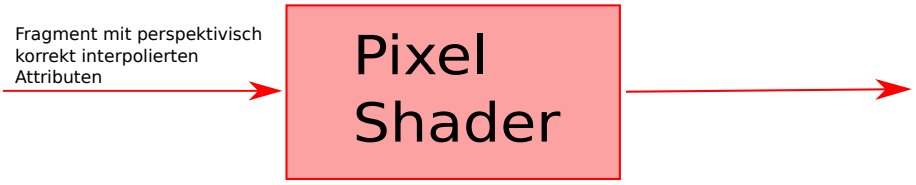

Abbildung 3.10: Pixel Shader

Die Ausgabe vom Vertex Shader, perspektivisch korrekt interpoliert über die Dreiecke, wird mit der Position des Pixels Input des Pixel Shaders. Eine minimalistische Implementierung des Pixel Shaders gibt die Fragmentfarbe zurück. Im darauffolgenden Schritt wird mit dem Z-Buffer auf Sichtbarkeit geprüft. In diesem Schritt zuvor können wir die Tiefen- und Alphwerte ändern. Man kann ganze Fragmente verwerfen(*discard*). Dies kann man für unterschiedliche Zwecke verwenden, u.a. kann man ganze Teile der Szenengeometrie ausschneiden. Die Ausgabe kann in mehrere verschiedene *render targets* geschrieben werden. Eine wichtige Vorraussetzung für *deffered shading*. Diese Form von Beleuchtungsberechnung erlaubt uns viele Lichtquellen gleichzeitig in der Szene zu benutzen ohne die signifikanten Leistungseinbußen wie beim *forwarding* zu haben. In einem ersten Schritt speichert man Informationen über das Material/Position vom Objekt in verschiedene *render targets*. In einem zweiten Durchlauf kann man nun die Beleuchtung und einige andere Effekte sehr effektiv berechnen. Das abschließende nicht komplett freiprogrammierbare, aber hoch konfigurierbare *Merging* hat eine besondere Aufgabe beim Abspeichern der Farbe für jeden Pixel im color buffer. Zur Bestimmung der aktuellen Farbe wird nun auch das Problem der Sichtbarkeit von Objekten angegangen. Zu den Z-Werten, welche wir als Tiefe beim Viewport Transform 3.2.7 gespeichert haben, gibt es hier Zugang zum Depth/Z-Buffer. Dieser Z-Buffer speichert anfangs überall den Wert inf. Beim Durchlauf der Geometrie wird nun jeweils für jeden Pixel, der die Geometrie bedeckt der color und depth buffer wie folgt aktualisiert: Ist der verglichene Tiefenwert des Fragmentes des Objekts kleiner als der Wert im Tiefenbuffer für den betroffenen Pixel, so schreibt er diesen Tiefenwert in den Z-Buffer und auch der color buffer mit der Fragmentfarbe aktualisiert. Anderseits passiert nichts und das nächste Primitiv wird betrachtet (Szene ohne semi-transparente Objekte!). Haben wir semitransparente Objekte, so müssen wir zuerst die Szene wie beschrieben ohne diese Primitive zeichnen, alle semi-transparenten Primitive nach ihrer Tiefe ordnen und in dieser Reihenfolge zu dem zuvor gerenderten Bild hinzufuegen. Damit haben wir auch unsere Projektion vollzogen, welche wir zuvor vorbereitet haben 3.2.5. Da der buffer bereits von dem vorherigen Durchlauf Farbwerte beinhaltet ist es nun daran eine Kombination aus aktuellen Inhalt, sowie einkommende Farbe zu finden, welche wiederrum im color buffer abgespeichert wird.

3.4 Per Fragment Operations

Auf den durch Rasterisierung entstandenen Fragmenten können noch einige Operationen vorgenommen werden.

3.4.1 Multisample Fragment Ops

3.4.2 Stencil Test

Damit können wir schlussendlich entscheiden, ob ein Fragment gezeichnet werden soll oder nicht durch den Vergleich mit einem Referenzwert und Stencil Buffer.(im Framebuffer).

3.4.3 Occlusion Query

3.4.4 Blending

Wir mischen Farbwerte die bereits in den Framebuffer geschrieben wurden mit den Farbwerten, welche in den erzeugten Fragmenten vorliegen. Die Art in der die beiden Farben gemischt werden bestimmt eine Funktion z.B Multiplikativ, Additiv...

3.4.5 Logical Operations

3.5 Koordinatensysteme

3.6 Compute Shader

Losgelöst von der Grafikpipeline steht der Compute Shader. Als freiprogrammierbare Einheit lässt sie sich nicht nur für graphische Berechnungen in der Pipeline, sondern auch für stark parallelisierbare Probleme im Allgemeinen einsetzen. Im Folgenden wird auf die DirextX12-API [Dir19,] eingegangen. Der Compute Shader arbeitet auf einer Threadgruppe(≤ 1024). In einem dreidimensionalen Array organisiert, teilen sich die Threads einer Gruppe Speicher, über den Sie kommunizieren und im Gesamten Fortschritt machen können, d.h. alle Threads laufen nebenläufig. Mit der zusätzlichen Eigenschaft auf Buffer Objekte (Daten auf der GPU) zugreifen zu können, haben die Compute Shader beispielsweise im Post-Processing 3.4 Bedeutung. Als Threadgruppe greifen sie jeweils auf einen bestimmten Bereich des Framebuffers zu und können die nötigen Nachbarschaftsinformationen austauschen.

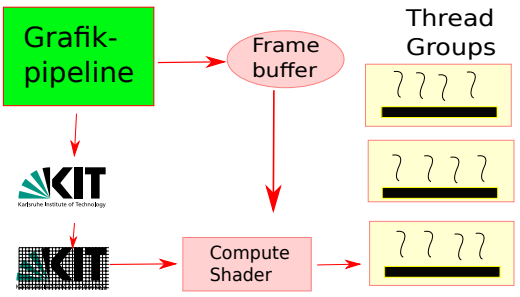

Abbildung 3.11: Funktionsweise Compute Shader

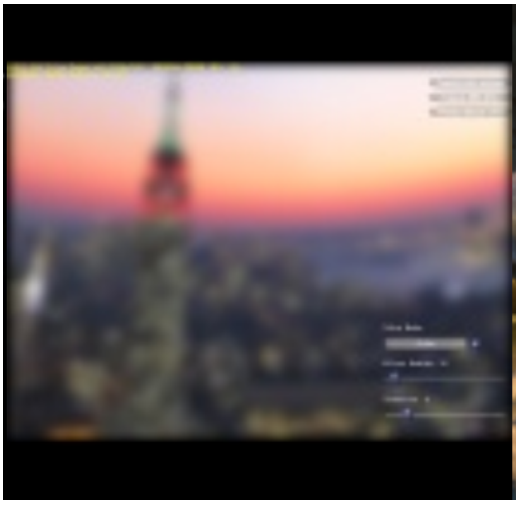

Abbildung 3.12: Postprocessing Blur mit Compute Shader [11]

Die erste Abbildung zeigt nochmal deutlich, dass der Compute Shader für jedwedige Art der Berechnung eingesetzt werden kann, bei dem *stream processors* zu tragen kommen. Eine Beobachtung war die effiziente Verlegung der Mesh Verarbeitung auf den Compute Shader. Dies führte zu der Einführung von Mesh Shadern 5.2, die stark an dem Aufbau von Compute Shadern orientiert sind.

4. Unterschied moderne und klassische Rendering-Pipeline

Schaut man sich die heutigen modernen Rendering-Pipelines(OpenGL 4.6, DirectX12) an, so kann man viele frei-konfigurierbare Stufen erkennen, die dem Entwickler viele Freiheiten in der Entwicklung geben. Frühere Konzepte wie z.B die OpenGL-States, welche die reine Auswahl von vorgegebene Zuständen vorgaben, haben sich als zu unflexibel in der Handhabung erwiesen. Eine limitierte Anzahl an Zuständen ohne freie Entwicklung eigener Konzepte wird den modernen Ansprüchen nicht mehr gerecht. So waren in früheren Versionen von OpenGL keine Beleuchtung pro Pixel(Phong Shading) möglich. Mehr Flexibilität auf Hardwareseite ermöglicht bessere Anpassungsfähigkeit an verschiedenen zu rendernde Szenenzuständen. Mit dem *unified shader* Ansatz gibt es keine Hardwareunterschiede zwischen Vertex- und Fragment Shader mehr. Flexibles Vergrößern/Verkleinern der Shaderpools ermöglicht ein Reaktion auf Geomtrie(viele Eckpunkte) lastige Szenen. Neue freiprogrammierbare Shadereinheiten waren auch Teil der Anwort auf den Wunsch nach mehr Freiheit und Flexibilität. So wurden der Tessellation und Geometry Shader (OpenGL 4.0, DirectX10) vorgestellt. Diese Einheiten waren aber auch eine Weitung des Flaschenhalses, der zwischen CPU und GPU entstand. Nachdem Grafikhardware in ihrer Leistungsfähigkeit deutlich schneller steigt als die der CPU bedeuten diese Einheiten eine Auslagerung von Arbeit auf die GPU. Ein Blick auf neueste Grafikhardware [nvi19] zeigt 14TFLOPS single precision, schnelleren GPU-Speicher (>600 GB/s) zu der gängigen Hauptspeicheranbindung (31 GB/s bei PCIe 4.0)[wik19]. Deswegen wird der frühere Immediate Mode vermieden um die Anzahl an Draw Calls zu reduzieren. Dafür kamen Vertex Array Obejekte, welche außerdem mit Index Chaching optimiert werden können.[CMFVH04] Sehr aktuelle Entwicklungen 5.2 zeigen eine erneute Steigerung der freien Konfiguerierbarkeit. Außerdem einen Einzug des Compute Shader 3.6 Modells in die Grafik-Pipeline. Ressourcen werden besser genutzt als beim Geometry Shader 3.2.4. Mehr dazu beim Ausblick 5. Eine weitere Neuerung zu der traditionellen Pipeline sind die Buffer Objekte.

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 3.6 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 "nähster Schnittpunkt"gefunden wurde. Hier kann der Programmierer die Methode *shade()* implementieren. Dabei sollte auf Schatten getestet, weitere Reflektions- und Transmission miteinberechnet werden
Intersection	Wird aufgerufen, falls wir eine Bounding Box unser Beschleunigungsstruktur treffen.
Miss	Wird aufgerufen, falls keine Szenengeometrie getroffen wurde.
Ray Generation	Mit *TraceRay()* können wir Strahlen verschießen. Normalerweiße 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+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 Reflektionen, 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

Der aktuelle Trend bei der Modellerstellung geht über zu immer komplexeren Modellen. Dabei stößt die traditionelle Pipeline mit der enormen Anzahl von Dreiecken an ihre Grenzen. Die damalige Einführung des Tessellation 3.2.3- und des Geometryshaders 3.2.4 (vorallen aber Tessellation!) war aus selbigen Grund und brachte eine erwünschte Weitung des Flaschenhalses und bessere Arbeitsaufteilung innerhalb der Pipeline. Das obige besprochene Primitive Assembly 3.2.1 erweist sich als *Fixed Function* innerhalb der Pipeline als zu star und unflexibel. Für starre Szenen wird hohe Bandbreite veranschlagt, da wir unseren Index Buffer jedesmal berechnen müssen. Anstatt wie beim Primitive Assembly 3.2.1 von der Hardware das Vertex Array Objekt aufs neue Berechnen zu lassen, teilt man nun die Geometrie selbst in *Meshlets* auf und speichert diese in einen Buffer(*Meshlet Desc Buffer*), wobei jedes *Meshlet* für sich auf einen VBO zeigt (Man hält den Buffer *On-Chip*; ähnlich Compute Shader Model 3.6). Diese Aufteilung gibt uns ein besseres Wiederverwenden der Eckpunkte und daher wird Bandbreite gespart! Die Mesh Shader kommen nach dem Compute Shader Model 3.6 und lassen eindeutig den Trend von fixen Vorgängen innerhalb der Pipeline zu mehr Flexibleren wiedererkennen. Durch diese freie Programmiermöglichkeit ergeben sich vielfältige Möglichkeiten, so z.B. für *Mesh compression* da sich die Buffer selber beschreiben lassen.

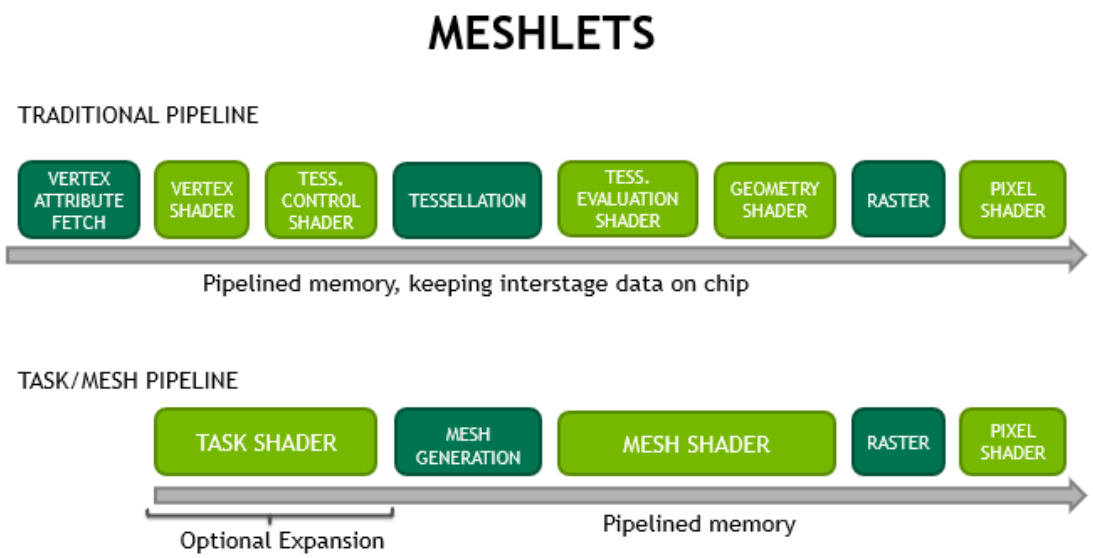

Abbildung 5.1: Vergleich bisherige Vertex,Tessellation,Geometry Pipeline <–> *stream procesor artig* Mesh Shader Pipeline

Dabei arbeitet der Task Shader ähnlich der Tessellation Control Einheit 3.2.3 und verteilt zunächst das Mesh in *workgroups*. Jedoch ist anzumerken, dass der Task Shader mit einem kooperativen Threadmodell arbeitet und dessen In-/Output komplett frei programmierbar ist. Man ist nicht mehr auf ein ankommendes Primitiv und eine ausgehende Tessellation Entscheidung beschränkt. Damit lassen sich Level of Detail Entscheidungen treffen, also wie fein unterteile ich mein einkommendes Primitiv. Oder man entscheidet sogar eine Gruppe gar nicht zu zeichnen (Culling) und schickt keine weitere Arbeit weiter. Der Mesh Shader nimmt lediglich die *workgroup id* entgegen und arbeitet ähnlich dem Compute Shader 3.6 auf einer Threadgruppe (hier langt jedoch ein eindimensionales Array). Es gibt hier jedoch nicht nur einen gemeinsamen Speicher, sondern auch einen Output, welcher frei-programmierbar ist. Zum Geometry Shader, der keine Threadgruppe besitzt, ist dies ein großer Schritt.

Literaturverzeichnis

[11] NVIDIA SDK 11. https://developer.nvidia.com/dx11-samples.

[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+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.

[Boy08] Chas Boyd: *The DirectX 11 compute shader*. ACM SIGGRAPH. Cité page, 25, 2008. http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf.

[CMFVH04] Howard H Cheng, Robert Moore, Farhad Fouladi und Timothy J Van Hook: *Vertex cache for 3D computer graphics*, apr 2004. US Patent 6,717,577.

[Dac18] Prof. Dr. Ing. Carsten Dachsbacher: *Vorlesung*. Internet, 2018. https://cg.ivd.kit.edu/lehre/ws2019/index_2114.php.

[Dir19] *DirectX12API*. Internet, May 2019. https://docs.microsoft.com/en-us/windows/desktop/direct3d12/direct3d-12-raytracing.

[Eng02] Wolfgang F Engel: *Direct3d Shaderx: Vertex and Pixel Shader Tips and Tricks with Cdrom*. Wordware Publishing Inc., 2002.

[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.

[nvi19] nvidia. Internet, May 2019. https://www.nvidia.com/de-de/geforce/graphics-cards/rtx-2080-ti/.

[tas19] *Mesh Shader*. Internet, May 2019. https://devblogs.nvidia.com/introduction-turing-mesh-shaders/.

[Vul19] *VulkanAPI*. Internet, May 2019. https://www.khronos.org/vulkan/.

[wik19] wikipedia. Internet, May 2019. https://de.wikipedia.org/wiki/PCI_Express.

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 28. Mai 2019

(Jonas Heinle)