Architektur und Programmierung von Grafik- und Koprozessoren

Die Grafik Pipeline

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

GPU Applikation - Operationsfluss

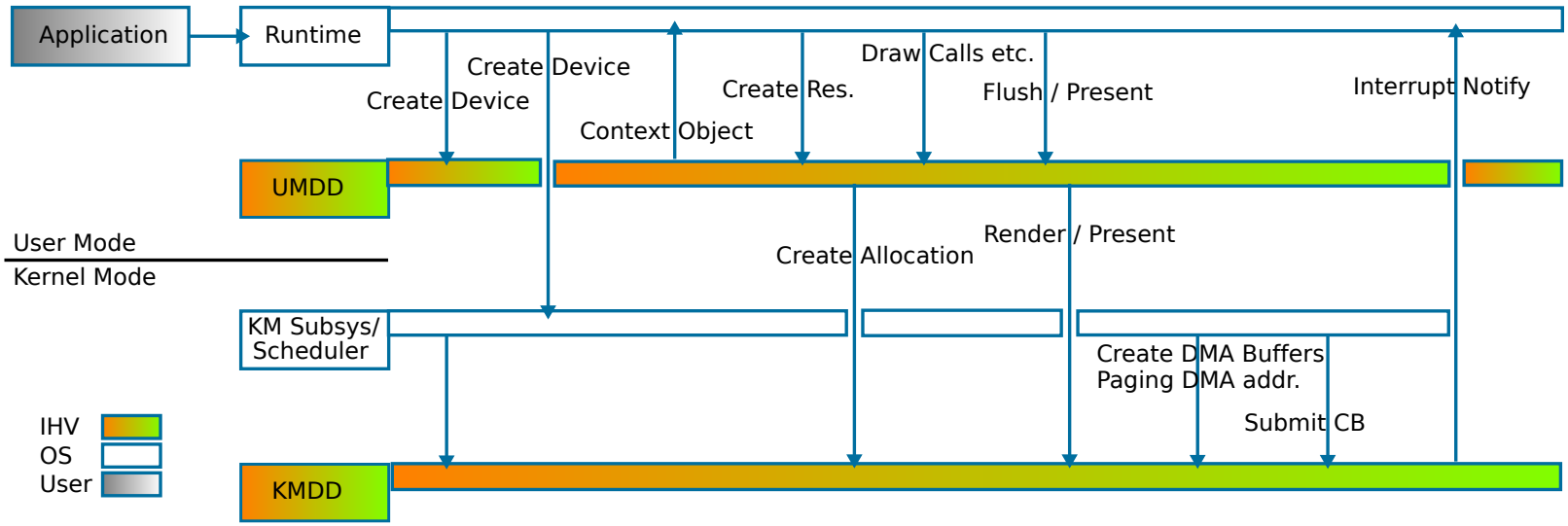


Abbildung: Darstellung (vereinfacht) gemäß Microsoft WDDM Design Guide: WDDM Operation Flow 14.03.2018.

GPU Applikation - Operationsfluss

Die Darstellung des Operationsflusses orientiert sich am WDDM, wird jedoch zum besseren Verständnis vereinfachend präsentiert.

1. Applikation stößt Rendering an, indem es ein *Rendering Device* mit *Rendering Kontext* bei Runtime anfordert.
   - Rendering Device abstrahiert HW Device, hat Eigenschaften, mittels derer man Leistungsfähigkeit, Speicherkapazität, etc. abfragen kann.
   - Rendering Kontext abstrahiert den Zustand der 3D Rendering Applikation.
2. User-Mode Treiber (UMDD) gibt `CreateDevice` Befehl an Kernel-Mode Subsystem weiter.
3. UMDD übergibt Rendering Kontext(e) an Runtime.

GPU Applikation - Operationsfluss

4. Ressourcenallokation (Vertex-Buffer, Texturen, etc.), wird von Runtime an UMDD, und von dort in Kernel Mode weitergereicht.
5. Zeichenbefehle und schlussendlich der Befehl zur Anzeige werden an UMDD übermittelt.
6. Kernel Mode Subsystem puffert Zeichen- und Anzeigebefehle in *kontextbezogenem Command Buffer* (CB), der mittels Direct Memory Access (DMA) direkt an die *GPU execution unit* übermittelt wird.
   - Die Datenpufferung erfolgt in einem dedizierten Speicherbereich, auf den nur die GPU per DMA zugreifen kann. Dieser muss für jeden CB erneut alloziert werden, da die physikalischen Adressen von Applikationen geteilt werden.
   - In den sequentiellen DMA CB wird ein Token gelegt (etwa ein ganzzahliges Handle), das als *Fence* zur Synchronisation dient.

GPU Applikation - Operationsfluss

7. Die GPU execution unit verarbeitet den CB. Findet die GPU execution unit das Fence Token, signalisiert es dem Kernel Mode Treiber (KMDD), dass der CB ausgeführt wurde.
8. KMDD signalisiert an User Mode Prozesse, dass gerendert wurde.

GPU Applikation - Operationsfluss

Bemerkung: GPU Readback

GPU Readback (d. h. Rendering in einen Offscreen Buffer und dann Transfer der Farb- und/oder Tiefeninformation in Host Speicher der Applikation) ist mit diesem asynchronen Modell besonders teuer: CPU und GPU müssen synchronisiert werden. Dazu muss die GPU zunächst den gesamten Command Buffer Inhalt abarbeiten. Der gesamte Parallelismus des Modells geht verloren:

- Zuerst leert die GPU den Command Buffer, währenddessen wartet die CPU.
- Dann muss die CPU den Command Buffer neu befüllen, währenddessen die GPU keine Arbeit zu verrichten hat.

GPU Applikation - Operationsfluss

Wichtig zu merken

- Modelle wie das *Windows Display Driver Model* existieren, um mehreren Applikationen geordnet Zugriff auf die Grafik Hardware zu gewähren.
- Wesentliche Komponenten zum Ansteuern der Grafik Hardware verteilen sich auf User Mode und Kernel Mode. Runtime APIs kommunizieren mit User Mode Bestandteil des Treibers. Generell: führe möglichst viele Operationen im User Mode durch.

GPU Applikation - Operationsfluss

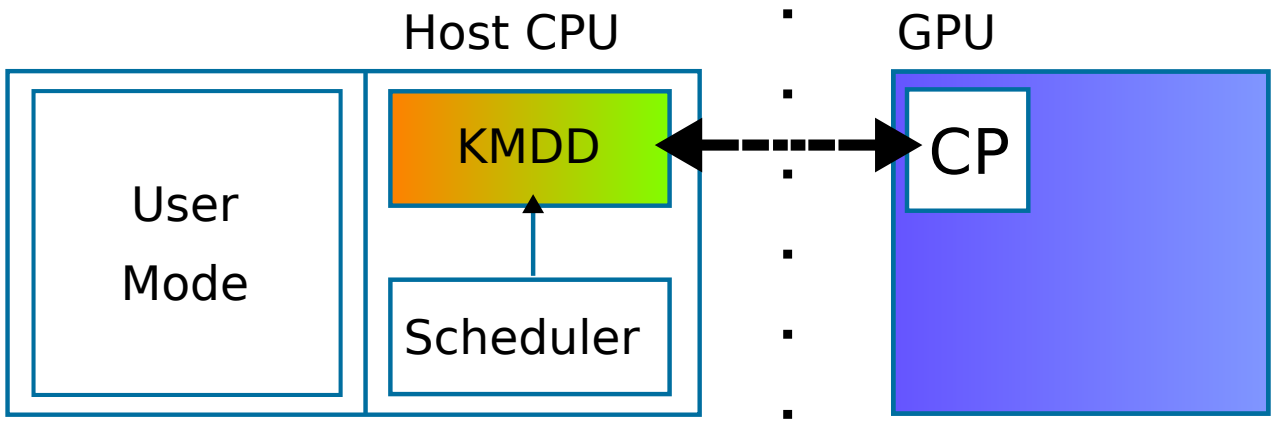
Wichtig zu merken

- Durch CBs können CPU und GPU asynchron kommunizieren. Die CPU kann zu jeder Zeit in den CB schreiben, GPU kann asynchron Command Packets aus dem Command Stream empfangen. Dazu ist Fence Synchronisation nötig.
- CBs werden in CPU Speicher aufgebaut, im priviligierten Modus und in eigenem Speicherbereich, auf den die GPU mittels DMA zugreifen kann.
- Da Grafikkarte geteilte Ressource, sind Synchronisationsmechanismen nötig, um die CBs Zeichenkontexten zuzuordnen und der Applikation zu signalisieren, dass gezeichnet wurde.

Der Command Processor

Command Processor

Command Processor (CP) steht am anderen Ende der eben beschriebenen Kommunikationskette und nimmt auf der Seite der GPU den Command Buffer entgegen.



CP hat die Aufgabe,

- den Kommandostrom aus CB entgegenzunehmen
- und die Kommandos vorzuanalysieren und an dedizierte Module auf der GPU (z. B. 2D, 3D) weiterzuleiten.

Command Processor

Push vs. Pull Modell

- CP verfügt i. d. R. über dedizierte *DMA Engines*, die auf dedizierten, priviligierten Host (=CPU) Speicherbereich zugreifen können.
- Push Modell: CB als Sequenz von *Command Packets*, meistens gepaart, mit "register writes", um den CP zu informieren, dass Daten ankommen. Host initiiert den CB Transfer.
- Pull Modell: CP fragt aktiv nach, ob neue CBs vorhanden und transferiert diese ggf. per DMA.
- GPUs unterstützen i. d. R. beide Modelle, zwischen denen man hin- und her schalten kann. Pull ist wegen DMA das präferierte Modell.

Command Processor

Skalierungsverhalten

- "Füllstand" des CBs:
  - Läuft der CB *leer*, ist Applikation CPU-limitiert, CPU kann nicht schnell genug neue Kommandos generieren.
  - Läuft der CB *voll*, ist Applikation GPU-limitiert, GPU kann Kommandos nicht schnell genug abarbeiten.
- Datensynchronisationspunkte: wenn CPU auf Ergebnisdaten von GPU zugreifen muss, muss synchronisiert werden.
- Besonders teuer: "Pixel-Readback" (lade das gerenderte Bild in CPU Speicher). CB muss erst komplett abgearbeitet werden, bis zum Readback werden keine neuen Kommandos eingeplant $\Rightarrow$ gesamte Pipeline läuft leer.

Command Processor

Ring Buffer

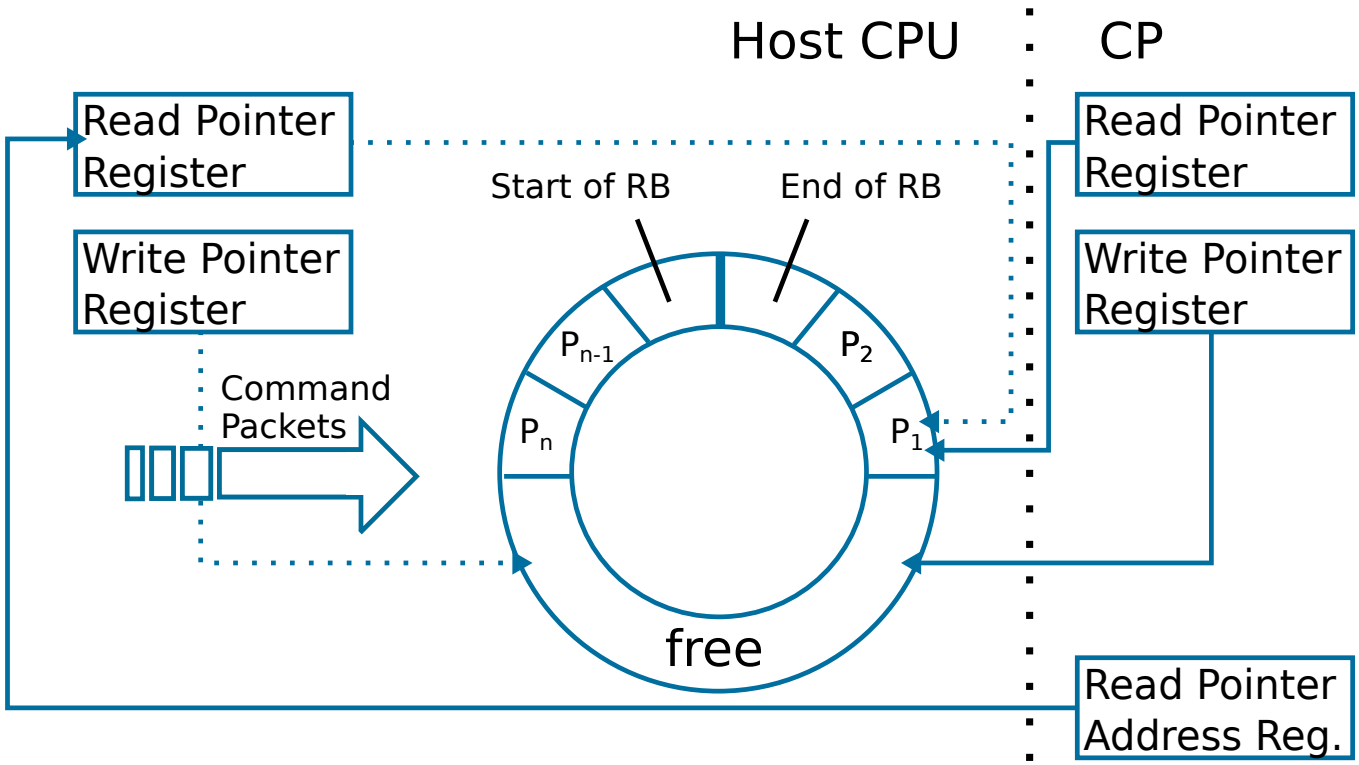


Abbildung: Darstellung (vereinfacht) gemäß AMD Dokument: Radeon R5xx Acceleration, Rev. 1.5, 08.06.2010.

Command Processor

Ring Buffer

- Ring Buffer eine mögliche Architektur; wird in Kombination mit Pull Modell verwendet.
- Ring Buffer in DMA Host Memory wird von Host mit Command Packets befüllt.
- Kommunikation erfolgt über Register auf Host und GPU, Host kann Register auf GPU direkt per “bus-mastering write” updaten.
- Häufig: Ring Buffer für 2D und 3D Zeichenkommandos, separate DMA Buffers (fast lane) für Datentransfer.
  - DMA braucht keine GPU Shader Ressourcen.
  - Keine virtuelle Addressauflösung beim Datentransfer.
  - Programmiermodelle wie Vulkan gehen im Grunde von DMA für Datentransfer aus.

Command Processor

Ring Buffer

- Host verwaltet den Ring Buffer:
  - verwaltet Zeiger auf Anfang und Ende, schreibt neue Command Packets in den noch freien Speicherbereich und stellt dabei sicher, dass der Ring Buffer niemals ganz voll wird.
  - updated Read Pointer direkt auf der GPU, wenn neue Pakete geschrieben wurden.
- CP
  - Greift per DMA über das Read Pointer Adressregister auf den Ring Buffer zu und
  - Nimmt Pakete einzeln aus dem Ring Buffer, bis dieser leer ist (Read Pointer = Write Pointer).

Command Processor

Command Packets

Exemplarisch für AMD Riva Architektur.

- Bestehen aus 32-bit Header und Body (n 32-bit DWORDs).
- Beispiele:
  - Type-0: schreibe N DWORDS in N aufeinander folgende Register
  - Type-3: Header: OP-Code zum Ausführen, Body: Daten
    - Beispiele:

`PAINT`	Zeichne N Rechtecke mit Füllfarbe
`BITBLT`	Kopiere Pixel von src nach dst ("blitting")
`POLYLINE`	Zeichne einen Linienzug
`WAIT_MEM`	Speichersynchronisation mittels Semaphore
`3D_LOAD_VBPNTR`	Lade Zeiger in Vertex Buffers
`INDX_BUFFER`	Lade Index Buffer
...	...

Command Processor

Command Packets

Command Buffer beinhaltet verschiedene Arten von Kommandos, u. a.:

- 2D Zeichenkommandos (werden i. d. R. an dedizierte 2D Hardware weitergereicht.
- Kommandos, die 3D Primitive an die Shader Pipeline übergeben.
- State Kommandos, um die Zustandsmaschine zu verändern.
- Kommandos für Speicherbewegungen.
- Shader Instruktionsfolgen (auch Compute).
- Uniforme Variablen.
- Synchronisationskommandos, z. B. um CPU und GPU zu synchronisieren, oder um Funktionseinheiten auf der GPU zu synchronisieren (mehr dazu später!).

Command Processor

Command Packets und Zustandsänderung

Denkmodell bei seriellen Architekturen:

```
static State global_state;

void func() {
    use_state(global_state);
    modify_state(global_state);
    use_state(global_state);
}
```

Dieses Denkmodell skaliert natürlich nicht, wenn `func()` von mehreren Threads gleichzeitig ausgeführt wird.

Command Processor

Command Packets und Zustandsänderung

Denkmodell bei Multi-Core Architekturen:

```
SHARED State global_state; // Shared Memory

void func() {
    use_state(global_state);
    LOCK();
        modify_state(global_state);
    UNLOCK();
    use_state(global_state);
}
```

Dieses Denkmodell skaliert natürlich nicht, wenn `func()` von *hunderten* Threads gleichzeitig ausgeführt wird.

Command Processor

Command Packets und Zustandsänderung

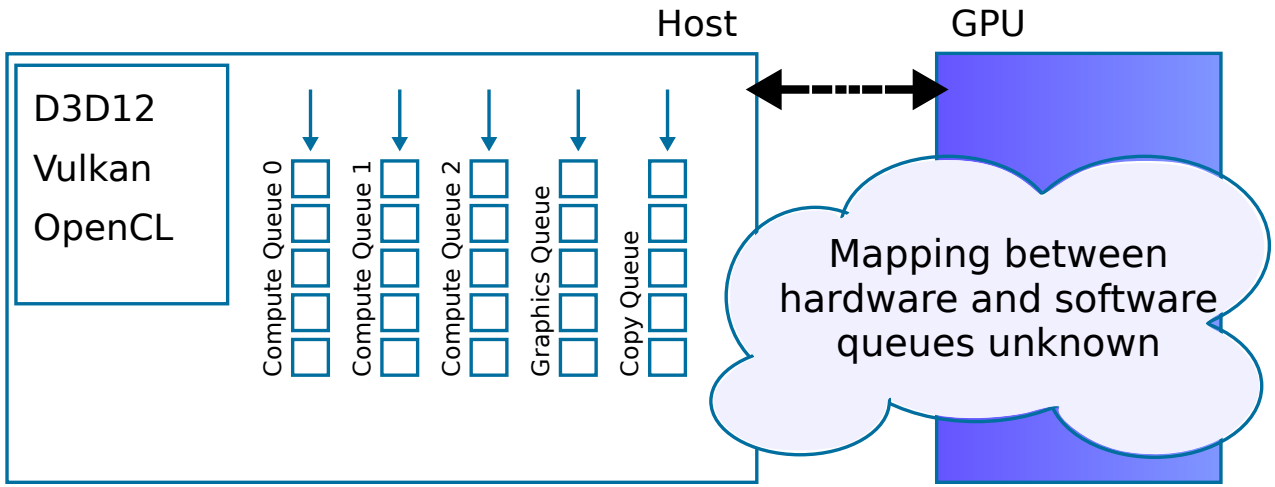
IHVs veröffentlichen wenig über tatsächliche Implementierungen.

Ein paar Anregungen:

- Einfachste Art, Zustandsveränderungen zu synchronisieren: “retained”: GPU propagiert Zustandsänderung, sobald alle Threads ihre Arbeit abgeschlossen haben.
  - Problem: Pipeline Stalls.
- Alternative: propagiere Zustand als Kommando durch die gesamte Pipeline. Ist bspw. die Fragment Stage am Materialzustand interessiert, ist das Zustandspaket direkt in der Nähe.
  - Problem: Skalierung, nur sinnvoll, wenn Zustand kompakt.
- Alternative: anstatt nur eines globalen States mehrere globale States; Änderungen “retained”. Dann muss nicht die ganze Pipeline angehalten werden.

Command Processor

Queues / engines etc. die vom API exponiert werden, mappen nicht notwendigerweise auf tatsächliche Hardware Queues.



Command Processors - AMD GCN

Beispiel AMD Graphics Core Next (z. B. Radeon R9 390X): Acht *Asynchronous Compute Engines* können über Queue 0-7 angesteuert werden. 3D Command Processor: Queue 0. Zusätzlich noch dedizierte DMA Buffer für Datentransfer.

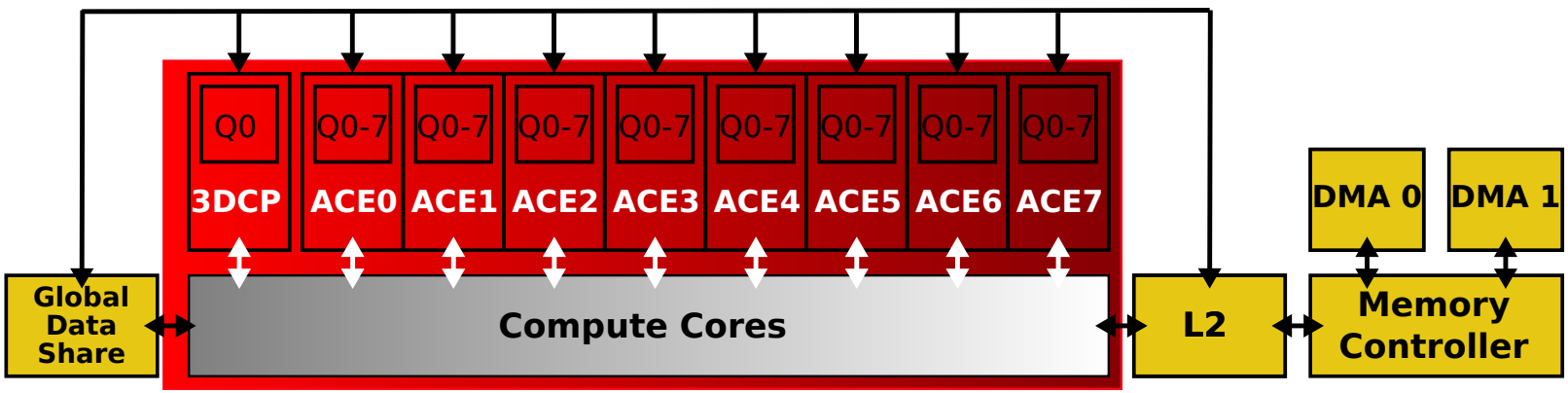


Abbildung: vgl. AMD Whitepaper: Asynchronous Shaders - Unlocking the full potential of the GPU.

API Order

- APIs haben eine strikte Regel: Zeichenkommandos müssen *dem Anschein nach* in der Reihenfolge ausgeführt werden, in der sie spezifiziert wurden ("API Order"). Damit einhergehend: die Reihenfolge, in der Dreiecke spezifiziert werden, determiniert, in welcher Reihenfolge sie gezeichnet werden.
- Große Hürde beim Design hochparalleler Architekturen.
- Design Entscheidungen bei Grafikkartenarchitekturen wesentlich durch API Order beeinflusst.

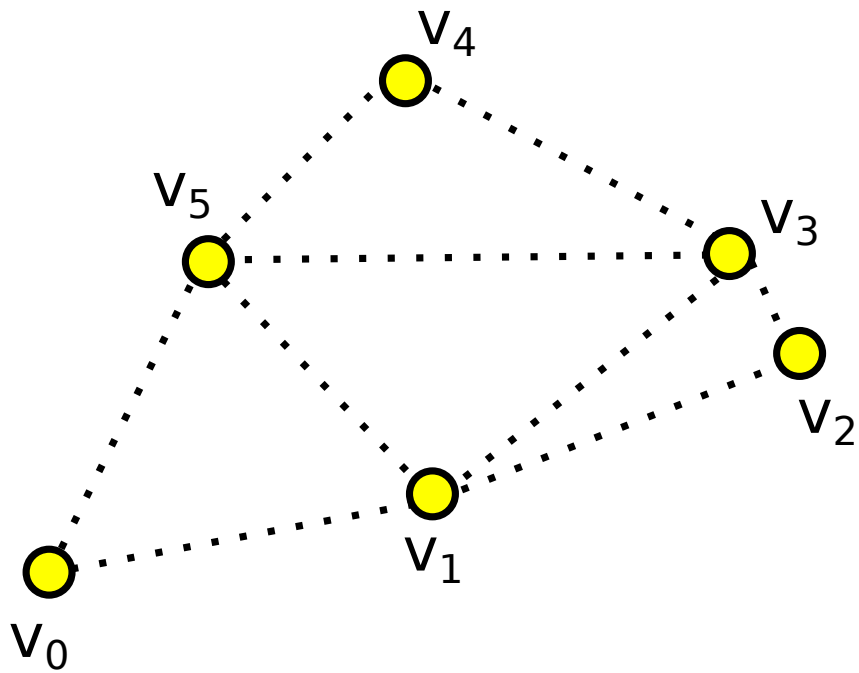
Vertex Phase

Vertex Phase

- Vertices werden transformiert. Kann einfache Matrix Transform sein, oder komplizierte Operation, die im Vertex Programm beschrieben wird.
- Vertex Attribute wie Normalen, Texturkoordinaten, Vertexfarben etc. werden ebenfalls transformiert.
- 1:1 Mapping: es verlassen so viele Vertices die Vertex Phase, wie hereinkommen (wir ignorieren Geometrie Shader, Tesselation Shader etc.).

Input Assembly

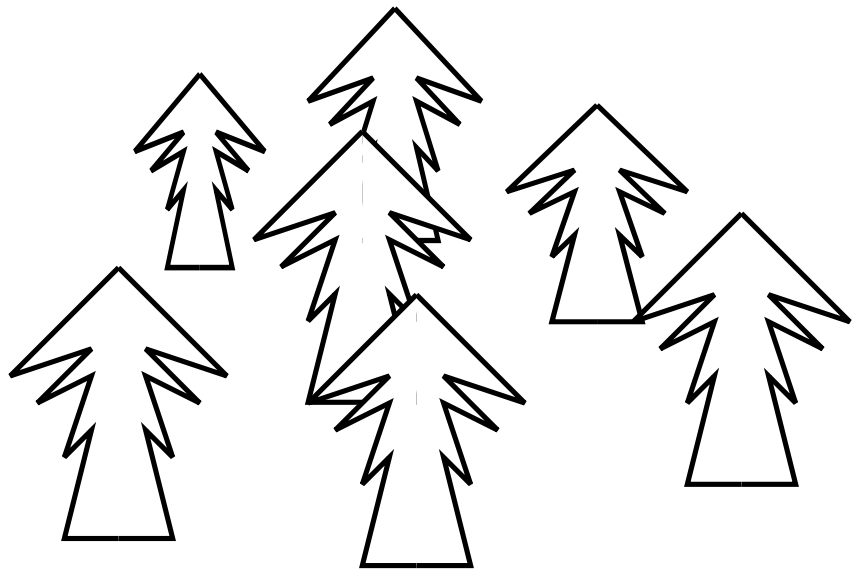
Indizierte Geometrie – Vertices werden üblicherweise von mehreren Dreiecken geteilt.




```
vertex_buffer_data(v0, v1, v2, v3, v4, v5, v6);
index_buffer_data({0,1,5}, {1,2,3}, {1,3,5}, {3,4,5});
```

Input Assembly

Instancing – instanziiere komplexe Geometrie, um Bandbreite zu sparen.



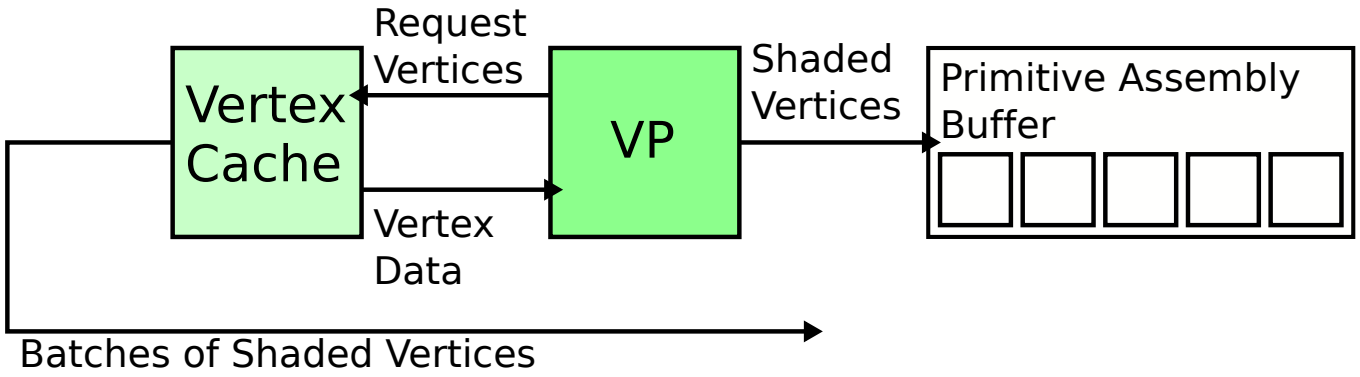
Input Assembly

- Input Assembly Stage: lese aus Index und Instance Buffers und reiche Vertices entsprechend an Vertexprozessoren weiter.
- Nicht so einfach: Vertex Strom nicht nur aus Speicher: da i. d. R. Vertices von 4-5 Dreiecken geteilt werden, werden transformierte Vertices in Cache gespeichert.
- Die sich anschließende Vertex Phase ist hochparallel: berücksichtigt keine Index Buffer, geteilte Vertices werden dupliziert. Index Buffer dienen im wesentlichen dazu, PCIe Bandbreite zu sparen.

Post Transform Cache

- Fixed-function: Cache mit fixer Anzahl an Vertices.
- Heute: Vertex kann mit variabler Anzahl Vertex Attributen ausgestattet sein $\Rightarrow$ je nach Anzahl Konfiguration passen mehr oder weniger Vertices in den Cache.
- Außerdem: Unified Shader sind für Durchsatz anstatt Latenz optimiert $\Rightarrow$ transformiere Batches von Vertices auf einmal (fixed-function: ein Vertex pro Vertex Shading Einheit). Vertex Cache entsprechend ausgelegt.

Vertex Phase



Post Transform Cache: nachdem Vertex Shader durchlaufen, werden geshadete und transformierte Vertices in den Cache geschrieben. Primitive Assembly Stage (nachfolgend) wird entweder aus Cache und durch noch zu transformierende Vertices bedient, die noch nicht im Cache stehen.

Vertex Caching komplex, da die Pipeline keine Konnektivität von Vertices speichert. Um zu bestimmen, ob zwei Vertices gleich, werden Vertex Attribute verglichen.

Vertex Phase

Vertex Phase (Beispiel Nvidia Fermi):[3]

- "Prozessor Cluster" (Gruppe von Cores) prozessieren Vertex Batches. 32 Threads ("Warp") pro Core.
- Threads verarbeiten Vertex Programm Instruktionen in lock-step. Wartende Threads (Branching) werden ausmaskiert (implizites Programmiermodell).
- Scheduler: Cores führen eine Warp auf einmal aus. Wartet eine Warp z. B. auf Speicheroperation, kann auf Core eine andere Warp geplant werden, die Arithmetik ausführt.
- Schnelles Umschalten zwischen Warps: jede Warp hat eigene Register im Register File $\Rightarrow$ Register knappe Ressource. Je mehr Register durch Shader alloziert, umso weniger Warps können gescheduled werden.

[3] vgl: Life of a triangle - NVIDIA's logical pipeline

Vertex Phase

Warp / Thread Group Scheduler

- z. B. Nvidia Kepler Architektur (GTX 680): 4 “Prozessor Cluster”, 192 Shader Cores pro Prozessor Cluster, 4 Warp Scheduler pro Prozessor Cluster.
- Instruktionen laufen bei Scheduler *in Gruppen* auf, Scheduler plant Warps möglichst sinnvoll.
- Warp führt eine Reihe (z. B. 2 oder 4) Instruktionen auf dem Shader Core aus, auf dem sie geplant wird.
- So kann Latenz versteckt werden: plane erst Warp, die aus Speicher liest; während diese wartet, plane Instruktionen von anderen Warps mit niedrigerer Latenz.
- Wegen Unified Shader Architekturen: Scheduling Logik für Vertex Phase und Fragment Phase gleich.

Vertex Phase

- Ausgabe fixed-function: Vertices in Normalized Device Coordinates.
  - Vertex Programme: Entwickler kann im Grunde selber entscheiden. Nach Vertex Phase jedoch Viewport Transformation (fixed-function)!
- Auf neuen Architekturen schließen sich noch weitere programmierbase Pipeline Stages an: Geometrie Stage, Hull & Domain Shader Programm. Diese generieren evtl. weitere Geometrie.
- Diese Pipeline Stages vernachlässigen wir bei unserer Betrachtung.

Primitive Assembly

Primitive Assembly

Bisher:

- Reine Verarbeitung auf Vertex Ebene, Konnektivität irrelevant.
- **Primitive Assembly**: führe Vertices mit Indizes aus *Index Buffer* zusammen.
  - GPU Primitive: Punkte, Linien, Dreiecke, Polygone.
  - Punkte: einfach.
  - Linien vernachlässigen wir, evtl. dedizierter Code Pfad.
  - Polygone auf Dreiecke abbildbar.
- **Culling / Clipping Phase**: jedes Dreieck wird am sichtbaren Frustum geclipped.
  - Einfache Dreiecke: vollständig innerhalb oder außerhalb Frustum.
  - Dreiecke, die Frustum *schneiden*, müssen geclipped werden.

Primitive Assembly

Idee 1

"Richtiger" Clipping Algorithmus, z. B. *Cohen-Sutherland*. Führe auf Dreiecken aus, generiert weitere Dreiecke.

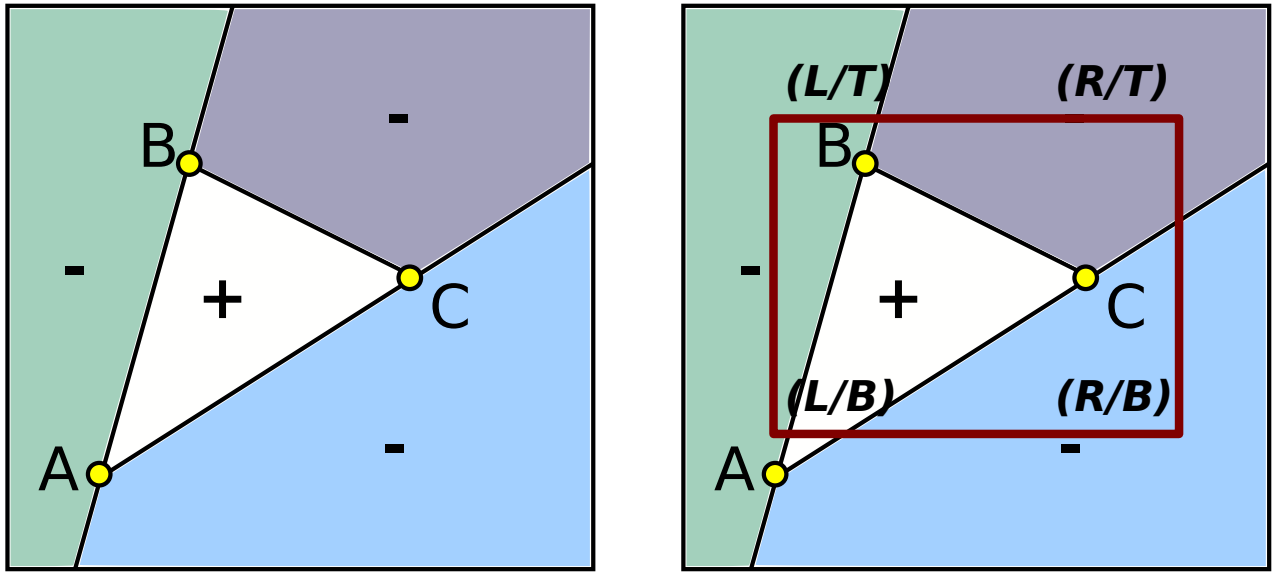
Idee 2

Wir erinnern uns an den Scan Konvertierungs Algorithmus aus Vorlesungsteil 2.

Anstatt weitere Dreiecke in die Pipeline einzufügen, identifizieren wir *konservativ* die Dreiecke, die echt außerhalb des Frustums liegen.

Anstatt zu clippen, führen wir Clipping Ebenen einfach als weitere *Kantenfunktionen* für den Scan Konvertierungs Algorithmus ein $\Rightarrow$ Clipping implizit durch Raster Engine, einfachere Hardware.

Implizites Clipping durch Raster Engines



- Anstatt zu clippen, übergebe weitere Kantenfunktionen an Raster Engines.
  - Dreieckskanten: $A - C$, $B - A$, $C - B$.
  - Frustum Kanten: $(L/T) - (L/B)$, $(R/T) - (L/T)$, $(R/B) - (R/T)$.
- Fensterkoordinaten: valide z-Werte zwischen 0 und 1 werden interpoliert. Clipping mit z-Near / z-Far trivial.

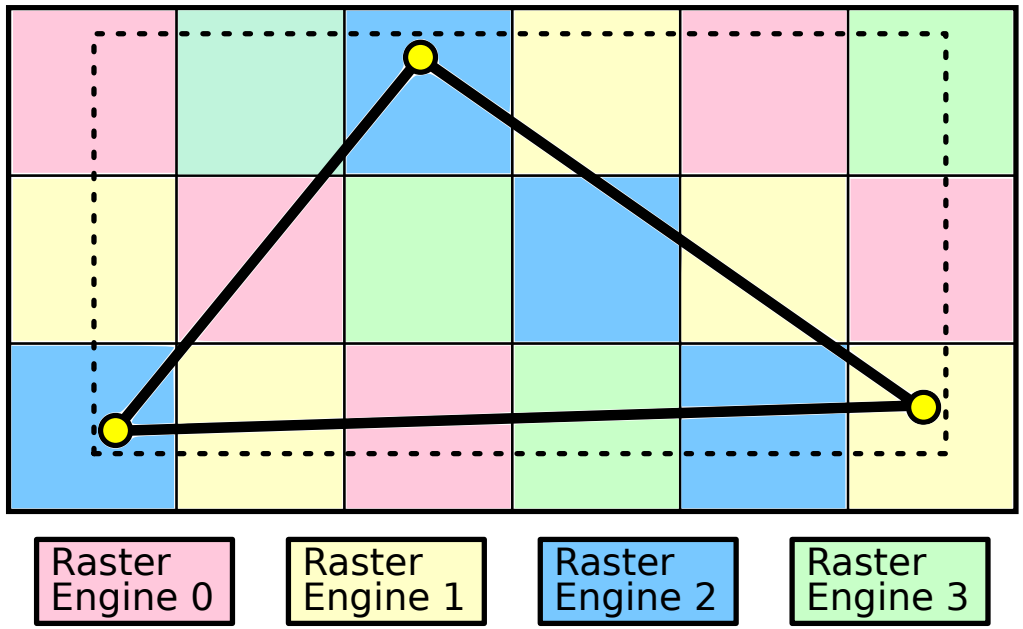
Scan Konvertierung

Scan Konvertierung

- Während dieser nicht-programmierbaren Phase werden aus Dreiecken Fragmente erzeugt.
- Es wird dann ein Fragment erzeugt, wenn der Mittelpunkt des Fragments ein Dreieck überlappt.
- Fragmente treten nach Scan Conversion immer in Vierergruppen ("Quads") auf.
- Bei der Scan Konvertierung werden Vertex Attribute *perspektivisch korrekt* über die Dreiecksinnenfläche interpoliert
- Falls aktiviert, wird hier Backface Culling durchgeführt.

Scan Konvertierung

Raster Engines



Mehrere Raster Engines führen Scan Konvertierung auf Kacheln aus (z. B. 8×8) und rastern *Quads* (vgl. Texture Filtering) in den Framebuffer / Speicherbereich für Compositing etc.

Größenordnung zwei, vier,.. Raster Engines auf GPU (vgl. Fast Tessellated Rendering on Fermi GF100 presentation @HPG'10).

Scan Konvertierung

Work Distribution Einheit

Bisher (Vertex Phase): eindeutige Zuordnung von Vertices / Primitiven zu Shader Prozessoren, keine Umverteilung.

Nun folgt erste Lastverteilungsphase, Primitive werden möglichst gleichmäßig auf Raster Engines verteilt.

Work Distribution Units (Nvidia: "Work Distribution Crossbar"): GPU Funktionseinheit, die Dreiecke (nach Primitive Assembly, in Fensterkoordinaten!) an Raster Engines verteilt.

Scan Konvertierung

Work Distribution Einheit

Viele moderne GPUs implementieren Sort-Last Algorithmus - Fragmente werden erst ganz zum Schluss in der Pipeline in API Order zurücksortiert.

Dazwischen findet Umverteilung auf verschiedene Funktionseinheiten statt: Raster Engines, ggf. Textureinheiten, Fragment Prozessoren.

GPUs gewährleisten API Order durch Tokens und durch Pufferung mit FIFOs.

Scan Konvertierung

Work Distribution Einheit

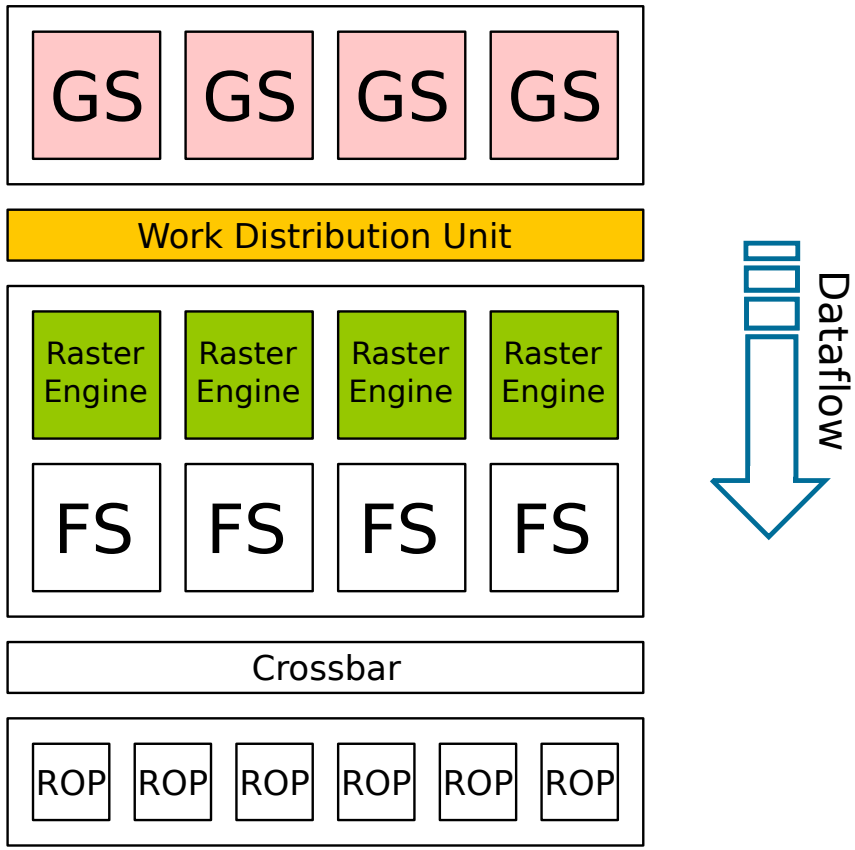
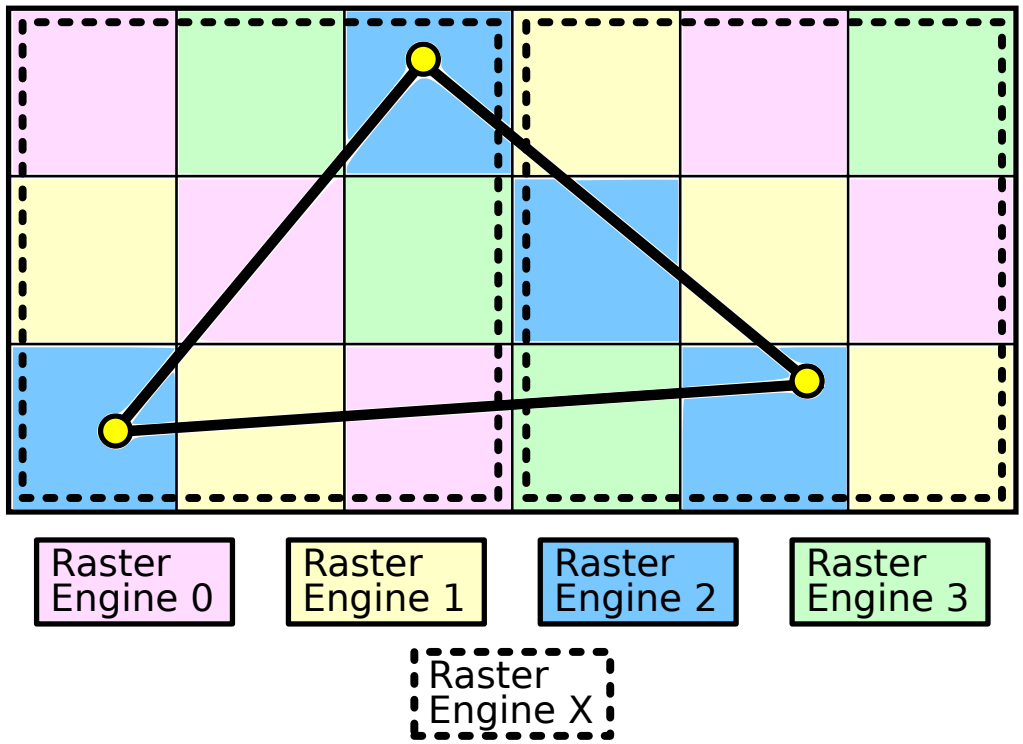


Abbildung: vereinfachend gemäß Tim Purcell: Fast Tessellated Rendering on Fermi GF100, HPG 2010. (GS = "Geometry Stage", FS = "Fragment Stage".)

Scan Konvertierung

Raster Engines



Mögliche Implementierung: flache Hierarchie - eine Raster Engine Stufe identifiziert $N \times N$ Kacheln, die tatsächlich gerastert werden müssen, weitere Raster Engine Stufen rastern Kacheln selbst und erzeugen Quads.

Scan Konvertierung

Raster Engines

- Kacheln: Lastverteilung in Screenspace. Kann zu starken Lastimbalancen führen, die sich kaum regulieren lassen (“Teapot in a Tile”).
- GPU Grundregel: vermeide kleine Dreiecke.

Scan Konvertierung

Quads und kleine Dreiecke

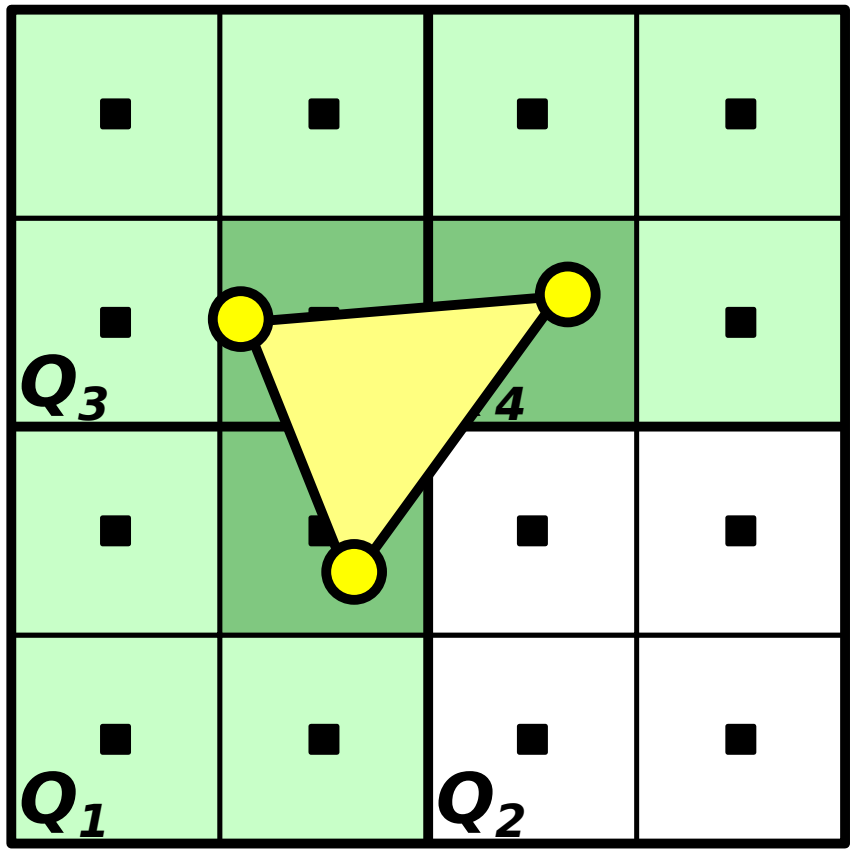


Abbildung: in Anlehung an Fabian Giesen: A trip through the graphics pipeline. Dreieck überdeckt nur drei Fragmente. Dennoch werden drei Quads mit insgesamt zwölf Einzelfragmenten erzeugt.

Scan Konvertierung

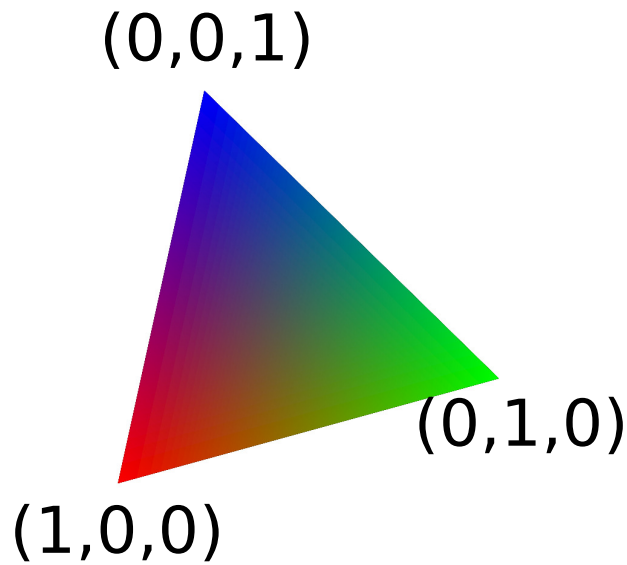
Triangle Setup

- Bestimme für jedes Dreieck die Kantenfunktionen für Scan Konvertierungsalgorithmus.
- Bestimme für jede Dreieckskante einen Referenzpunkt und rechne Wert für Kantenfunktionen dafür aus. Speichere Werte mit Dreieck.
- Raster Engine muss nun nur Additionen durchführen, um Coverage zu bestimmen.

Scan Konvertierung

Interpolation

- Interpoliere Vertexattribute (Tiefe, Normalen, Texturkoordinaten, etc.) über Dreiecksfläche.
- Wichtig: interpoliere perspektivisch korrekt $\Rightarrow$ baryzentrische Koordinaten.
- Baryzentrische Koordinaten trivial durch Kantenfunktionen bestimmbar.



Early z-Test

"The OpenGL Specification states that these operations happens after fragment processing. However, a specification only defines apparent behavior, so the implementation is only required to behave "as if" it happened afterwards.

Therefore, an implementation is free to apply early fragment tests if the Fragment Shader being used does not do anything that would impact the results of those tests."[4]

[4] `https://www.khronos.org/opengl/wiki/Early_Fragment_Test` (OpenGL Dokumentation)

Early z-Test

- Wenn immer möglich, rejecte Fragment, *bevor* wir es aufwendig shaden.
- Dafür werden die Instruktionen im kompilierten Fragment Shader (z. B. vom Treiber) voranalysiert, bevor der Fragment Shader ausgeführt wird.
- GPU führt early-z Optimierung durch, wann immer möglich. Kontraproduktiv:
  - Fragment Shader modifiziert Tiefe des Fragments
  - Fragment Shader ruft `discard()` für manche Fragmente auf
    $\Rightarrow$ diese werden nicht weiter betrachtet (dann können Fragmente dahinter aber nicht "early-z rejected" werden).

Early z-Test

Weitere Optimierung: Raster Engines bearbeiten Kacheln. Tiefenwerte werden über Dreiecke interpoliert. Bestimme einfach die interpolierten Tiefenwerte an den Kachelecken und *culle* wenn möglich die ganze Kachel.