Architektur und Programmierung von Grafik- und Koprozessoren

Die Grafik Pipeline

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

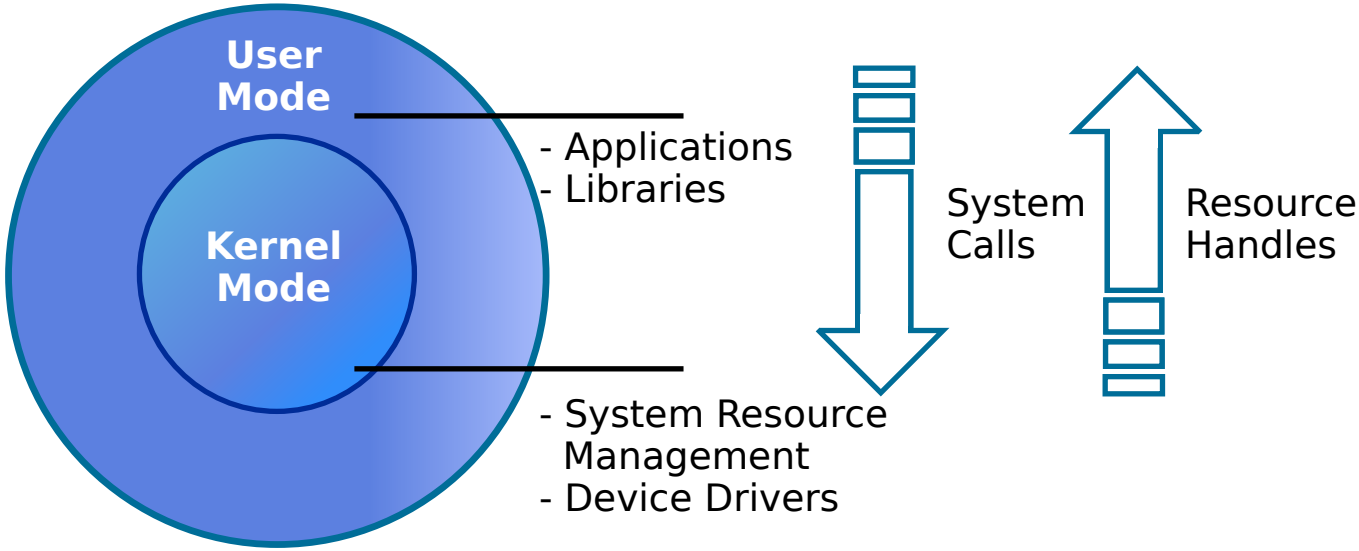
Host CPU und Grafiktreiber

Host Interface

- Applikation submittiert Vertices, Zeichenbefehle, State etc.
- Geht in Form von *Kommandos* durch das "Host-Interface":
  - Runtime und Treiber auf Seiten der CPU.
  - Command Processor auf Seiten der GPU.

Betriebssysteme - Protected Mode

Operationsmodi von CPUs und Betriebssystemen



Betriebssysteme - Protected Mode

- CPUs regulieren den Zugriff auf Systemressourcen (Speicher, Netzwerk, Geräte, etc.) mittels einer Ringstruktur (bei x86 Systemen: Ring 0 und Ring 3).
- Software, die im inneren Ring (Ring 0) ausgeführt wird, ist bzgl. Ressourcenzugriff hochpriviligiert ("Kernel Mode"), Software im äußeren Ring ist niedrigprivilegiert.
- API für Ressourcenzugriff: *System calls*. Applikation fragt beim Kernel mittels system call Ressource an und bekommt, so die Ressource verfügbar ist, ein *Ressourcen Handle*. Über das Ressourcen Handle wird die Ressource verwendet und freigegeben.
- Alle Operationen, die Kommunikation mit dem Kernel erfordern, sind vergleichsweise zeitintensiv.

Betriebssysteme - Protected Mode

Beispiel Dateisystemzugriff

C-Library Aufruf (vgl. `man 3 fopen`, `man 3 fclose`).

```
/* fopen.c */
#include <stdio.h>
int main() {
    FILE* fp = fopen("/home/zellmans/fopen.c", "r+w");
    fclose(fp);
}
```

System Call (vgl. `man 2 open`, `man 2 close`), Output mit `strace`.

```
...
open("/home/zellmans/fopen.c", O_RDWR) = 3
close(3)                                = 0
exit_group(0)                           = ?
++ exited with 0 +++
```

Betriebssysteme - Protected Mode

- Monolithisches Kernel Modell: Betriebssystem läuft im Protected Mode. Funktionalität wird duch *Kernel Module* ergänzt, die zur Laufzeit geladen oder ausgestoßen werden können.
  - Viele moderne Betriebssyteme wie Linux oder BSD basieren auf dem monolithischen Kernel Modell.
- *Gerätetreiber* werden als Kernel Module realisiert, die auf Ressourcen wie externe Hardware zugreifen können. Treiber i. Allg. sind Programme, die sich auf User Mode und Kernel Mode verteilen dürfen (in diesem Fall müssen die jeweiligen Treiber Bestandteile als separate Betriebssystemprozesse ausgeführt werden).

Grafiktreiber - Kernel und User Mode

Betriebssysteme definieren Schnittstelle, die Grafikkartenhersteller (*independent hardware vendor* (IHV)) implementieren müssen, z. B. *Windows Display Driver Model* (WDDM).

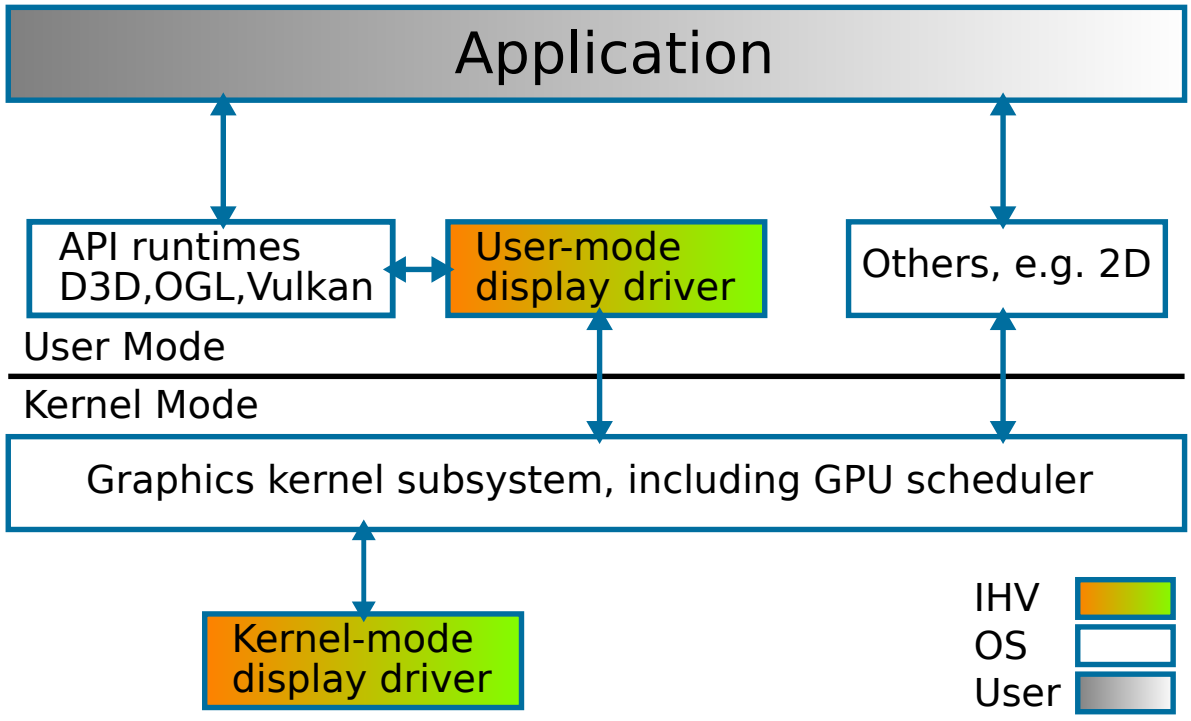


Abbildung: Darstellung (vereinfacht) gemäß Microsoft WDDM Design Guide: WDDM Architecture 14.03.2018.

Grafiktreiber - Kernel und User Mode

- Microsoft WDDM exemplarisch, andere OSs ähnlich.
- IHVs stellen Kernel- und User Mode Treiberkomponenten bereit, GPU scheduler kommt vom OS.
- User Mode Treiber übernimmt Aufgaben wie Patchen und Übersetzen von Shader Programmen in Intermediär Code (PTX, AMD IL) etc.
  - bei modernen APIs wie Vulkan oder D3D12 wird per Default keine Runtime Shader Compilation mehr durchgeführt.
- Prinzip: Verlagerung von möglichst vielen Aufgaben in User Mode verhindert, dass einzelne Programme das gesamte Grafik Subsystem zum Absturz oder Einfrieren bringen.

Grafiktreiber - Kernel und User Mode

- Virtualisierung: User Mode Prozesse können nicht auf den Grafikspeicher anderer Prozesse zugreifen. Virtualisierung in User Mode, nicht im Treiber.
- Prozess-Scheduling und Command Preemption durch User Mode Runtime.

GPU Applikation - Operationsfluss

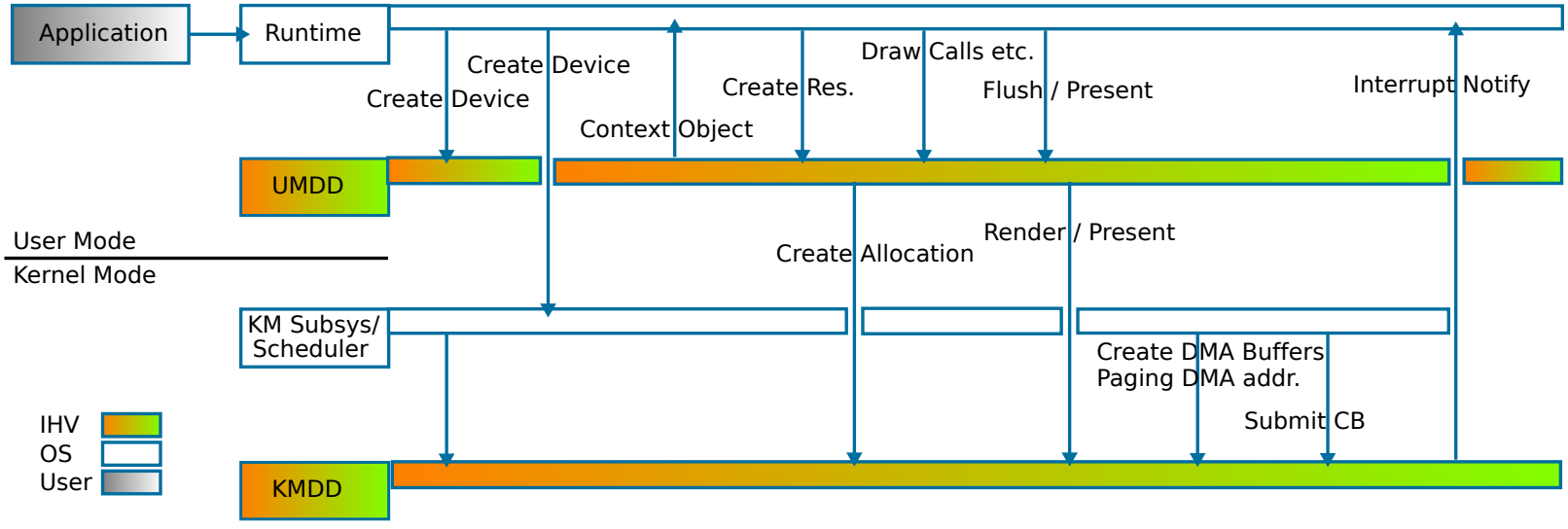


Abbildung: Darstellung (vereinfacht) gemäß Microsoft WDDM Design Guide: WDDM Operation Flow 14.03.2018.

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

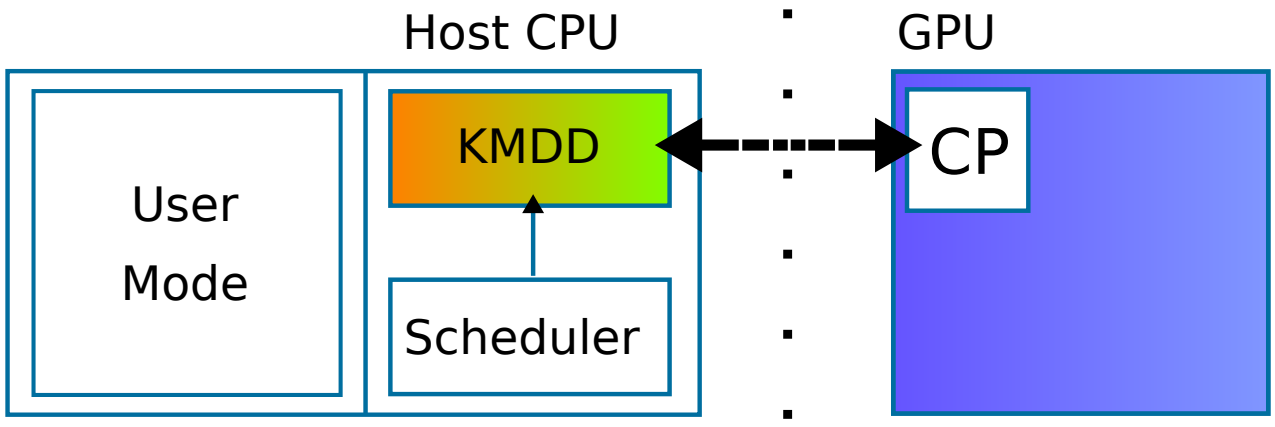
Wichtig zu merken

- Durch CBs können CPU und GPU asynchron kommunizieren. Dies ist eine Art von Nebenläufigkeit. 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

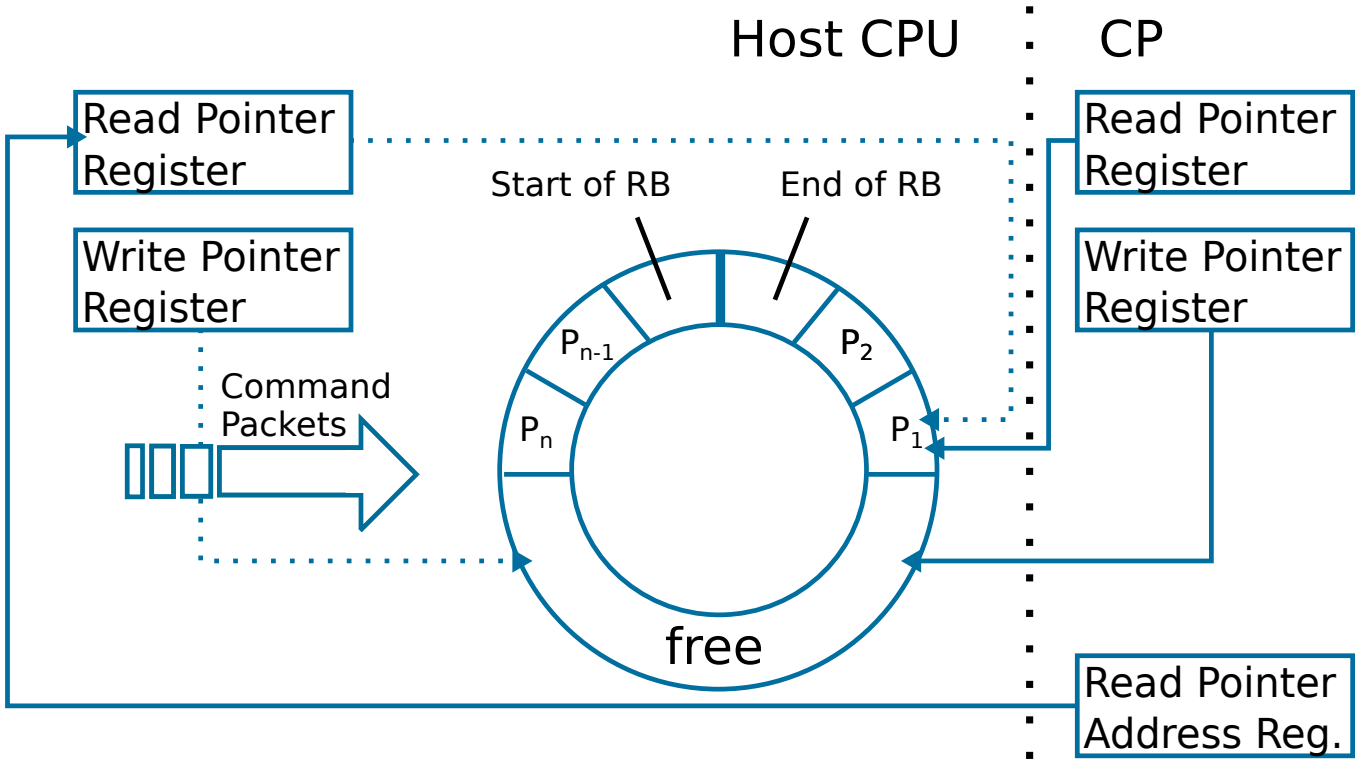


Abbildung: Darstellung (vereinfacht) gemäß AMD Dokument: Radeon R5xx Acceleration, Rev. 1.5, 08.06.2010.

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:

```
thread_local 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.