Übung: Interaktive Computergrafik

Ablauf

- Übungsblätter
  - Meistens wöchentlich
  - Freiwillig
  - Besprechung der Lösung in der Übung (Mi, 10:30 – 11:15)
  - “Sprechstunde” für Fragen in der ATIS (Mi, 9:45 – 10:30)

- Übungspapers
  - Meistens wöchentlich
  - Diskussion in der Übung (Mi, 10:30 – 11:15)

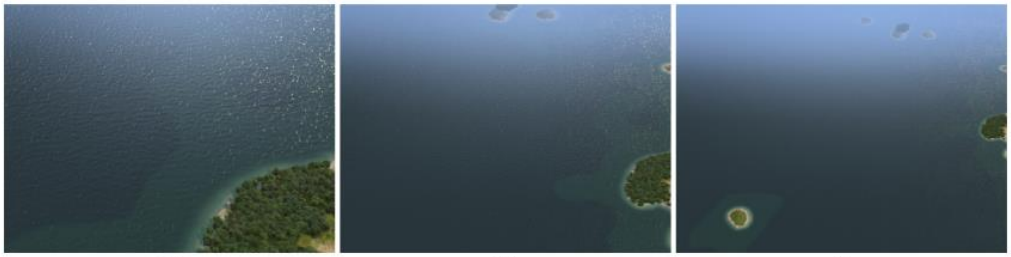
- Je nach Zeit: Wissenswertes zum Stand der Technik

Erstes Übungspaper

LEAN Mapping

Marc Olano*
Firaxis Games

Dan Baker†
Firaxis Games



(a) (b) (c)

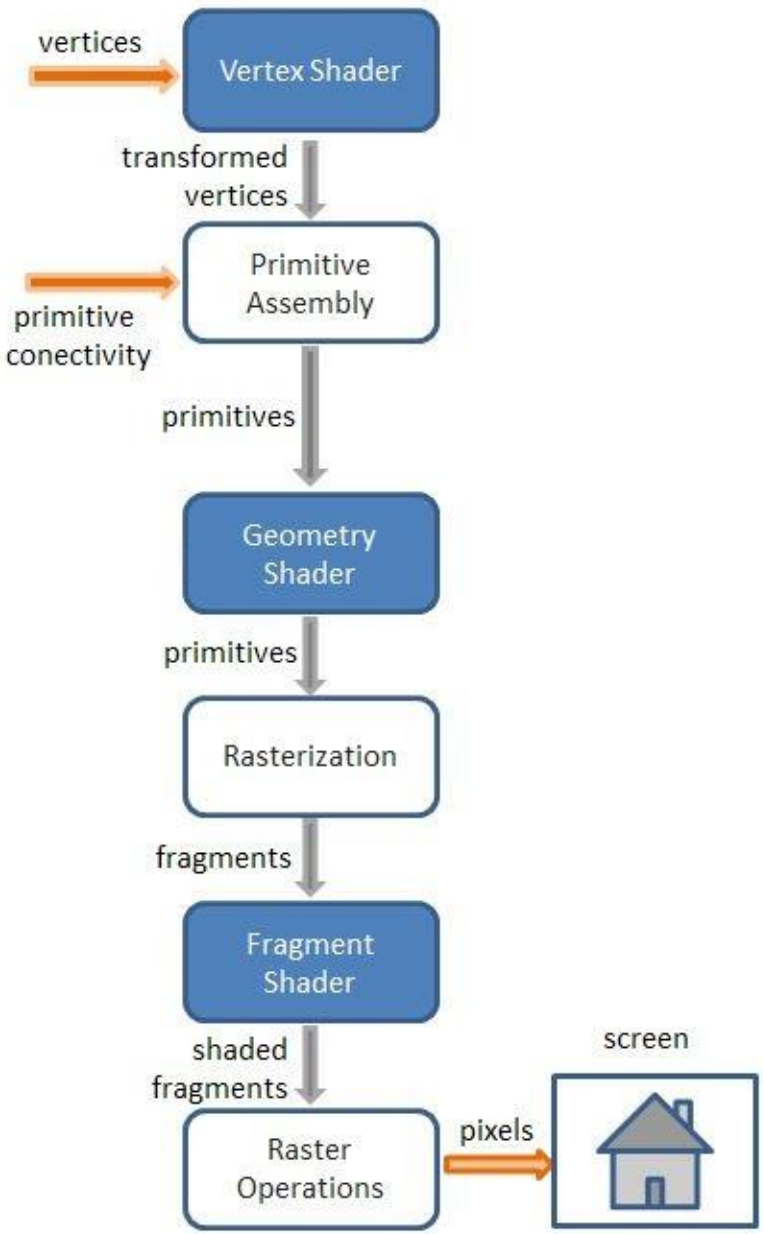
Figure 1: *In-game views of a two-layer LEAN map ocean with sun just off screen to the right, and artist-selected shininess equivalent to a Blinn-Phong specular exponent of 13,777: (a) near, (b) mid, and (c) far. Note the lack of aliasing, even with an extremely high power.*

Marc Olano and Dan Baker: LEAN Mapping
http://www.csee.umbc.edu/~olano/papers/lean/

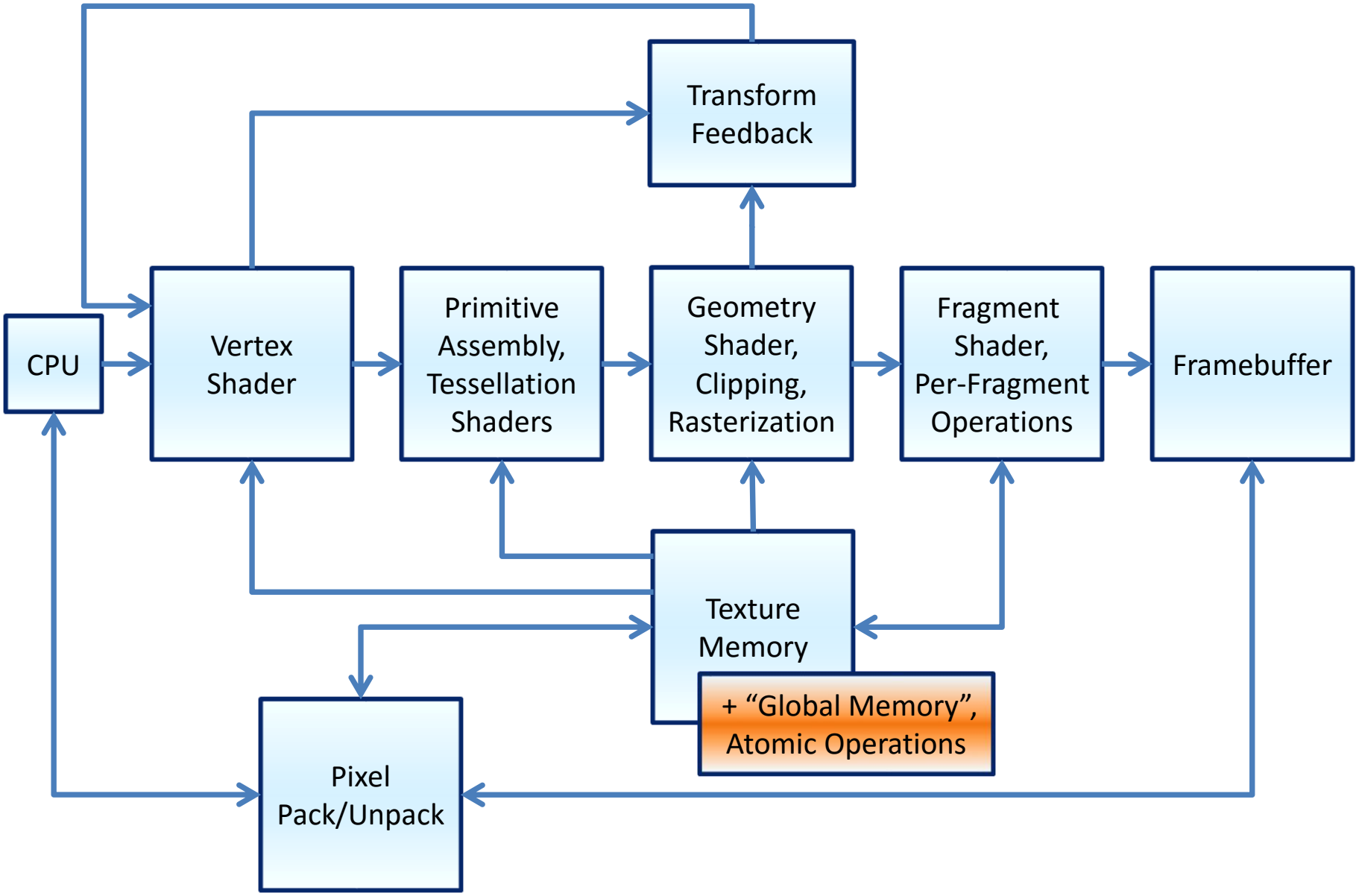
Heute

- Hardware-beschleunigtes Rendering
  - Kurze Wiederholung APIs
  - Was machen GPUs daraus?

Moderne OpenGL Pipeline – 3.x



Moderne OpenGL Pipeline – 4.x



Modernes OpenGL

- Shader-Stufen sind programmierbar
  (Shader heute in etwa synonym mit „Pipeline-Callback“)

- Vertex-, Fragment-, Geometry-, Tessellation- und Compute Shaders
  - Typ bestimmt Schnittstelle (Eingaben, Ausgaben)

- C-ähnliche Programmiersprachen (GLSL, HLSL, ...)
  - Eigenen Convenience-Datentypen (vec3, mat4, ..., vgl. GLSL Spec/glm)

- Mittlerweile Random Read-Write Access in Buffers und Texturen
  - Read-only ggf. effizienter (Ping pong vs. Inplace, Storage vs. Overhead)
  - Effizienz *stark* abhängig von Cache-freundlichen Zugriffsmustern

- Flexible Ressourcenverwaltung (Speicherort, Datentransfer)
  - Schneller VRAM auf GPU, derzeit 4-12 GiB
  - Asynchrone Datenaktualisierung, Verstecken von Übertragungslatenzen

Geometrieverarbeitung: Vertex Shader

- Transformation **einzelner** Vertices und Verarbeitung deren Attribute
  - keine Vertex-Erzeugung oder -Löschung
  - keine Informationen über andere Vertices
- Berechnung von Attributen, die für Fragmente interpoliert werden sollen
  - z.B. Beleuchtung per Vertex (Gouraud Shading, altmodisch) oder
  - Normalen für Phong Shading

Vertex Shader

- Eingaben sind üblicherweise per Vertex
  - Position
  - Normale
  - Farbe oder Texturkoordinate
- Ausgabe
  - Weitergeleitete Vertex-Attribute
  - Position nach MVP-Transformation

- Vorstellung
  - Für jeden Vertex wird parallel ein Vertex Shader von der Grafik-HW ausgeführt
  - „Ein Thread pro Vertex“

Vertex Shader

- Beispiel:

```
layout(location = 0) in vec3 POSITION;
layout(location = 1) in vec3 NORMAL;

out vec3 world_position;
out vec3 world_normal_interpolated;

void main() {
  gl_Position = MVP * vec4(POSITION, 1.0);

  world_position = vec3(M * vec4(POSITION, 1.0));
  world_normal_interpolated = N * NORMAL;
}
```

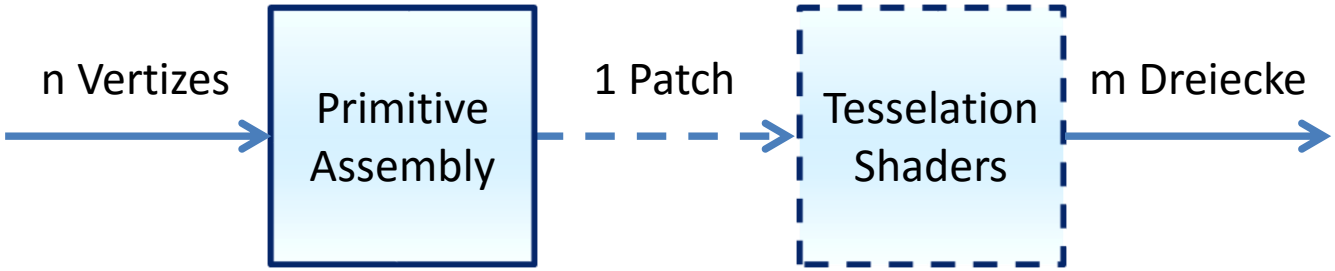
- **POSITION** und **NORMAL** werden beim draw-Aufruf an die Grafik-HW geschickt
- Wie kann man die Matrizen **MVP**, **M** und **N** übergeben?

Uniforms

- Können durch OpenGL Aufrufe mit Daten gefüllt werden
- Bleiben während des gesamten draw-Aufrufs konstant

```
uniform mat4 MVP;
uniform mat4 M; // from model to world space
uniform mat3 N; // from model to world space
                // for normals

layout(location = 0) in vec3 POSITION;
layout(location = 1) in vec3 NORMAL;

out vec3 world_position;
out vec3 world_normal_interpolated;

void main() {
  gl_Position = MVP * vec4(POSITION, 1.0);

  world_position = vec3(M * vec4(POSITION, 1.0));
  world_normal_interpolated = N * NORMAL;
}
```

Geometrieverarbeitung: Assembly + Tesselation



- **Primitive Assembly** setzt aus den Vertices die angeforderten Primitive zusammen und ist **nicht programmierbar**:
  - `GL_POINT, GL_LINE[*], GL_TRIANGLE[*]` in ihren unterschiedlichen Modi `*_FAN, *_LOOP, *_STRIP`
- **Tesselation Shaders**
  - Tesselation Control Shader bestimmt die Unterteilung
  - Tesselator führt sie durch (nicht programmierbar)
  - Tesselation Evaluation Shader arbeitet auf der resultierenden Geometrie
  - spezieller Primitivtyp `GL_PATCH`

Beispiel Tessellation – Unigine Benchmark

Eingabegeometrie vor der Unterteilung durch die Tesselationseinheit

Beispiel Tessellation – Unigine Benchmark

Displacement Mapping: Unterteilung und Verschiebung der neuen Vertizes

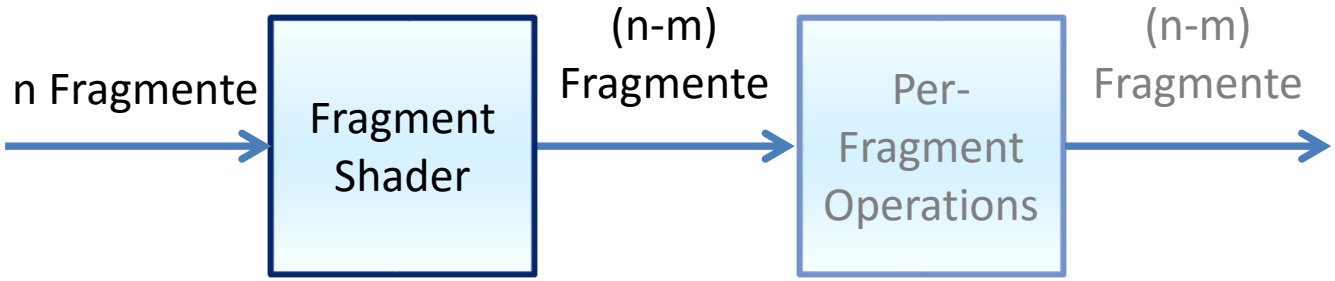
Geometrieverarbeitung: Geometry Shader etc.

- Geometry Shader (programmierbar, aber optional)
  - bearbeitet (parallel) einzelne Primitive (Punkt, Linie, Dreieck)
  - kann Primitive vervielfachen, entfernen oder beliebig umwandeln (Punkte zu Dreiecke etc.)
- Beispiele
  - Expansion von Punkten zu rechteckigen Zeichenflächen für Partikel
  - Berechnung von Dreiecksattributen (Normalen, Tangenten ...)
  - Extrudieren von Dreiecken zu Hüllkörpern

Geometrieverarbeitung: Geometry Shader etc.

- Geometry Shader (programmierbar, aber optional)
  - bearbeitet (parallel) einzelne Primitive (Punkt, Linie, Dreieck)
  - kann Primitive vervielfachen, entfernen oder beliebig umwandeln (Punkte zu Dreiecke etc.)
- Clipping (am Sichtvolumen) ist nicht programmierbar
  - verwirft unsichtbare Punktprimitive
  - verändert Dreiecke durch Schnitt mit Sichtvolumen
- anschließend: perspektivische Division (nicht programmierbar)
- Rasterisierung generiert Fragmente für den Ausgabepuffer…

Fragmentverarbeitung



- …für jedes dieser Fragmente wird ein **Fragment Shader/Program** ausgeführt
  - Berechnung von **Farbe (RGB+Alpha)** und optional **Tiefenwert** pro Fragment
    - z.B. Phong Shading, also Beleuchtungsberechnung pro Pixel
  - Eingabe-Attribute werden innerhalb des Primitivs interpoliert
    - z.B. Vertex-Normalen, die vom Vertex Shader weitergegeben wurden

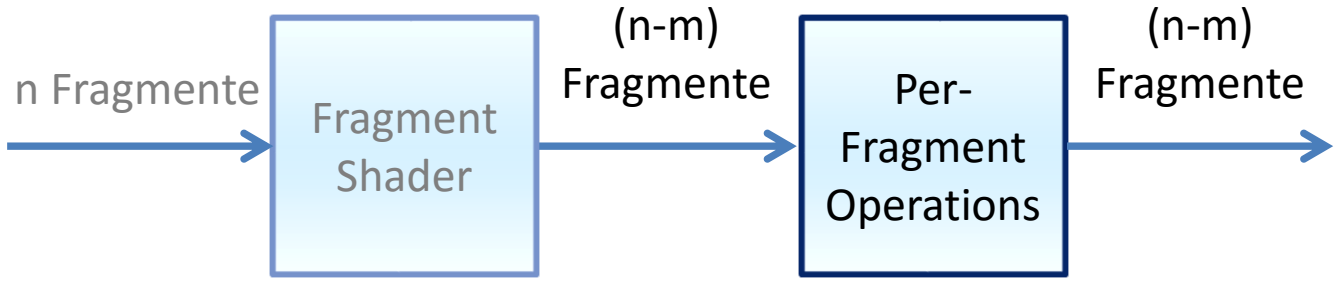
Fragment Shader

- Beispiel:

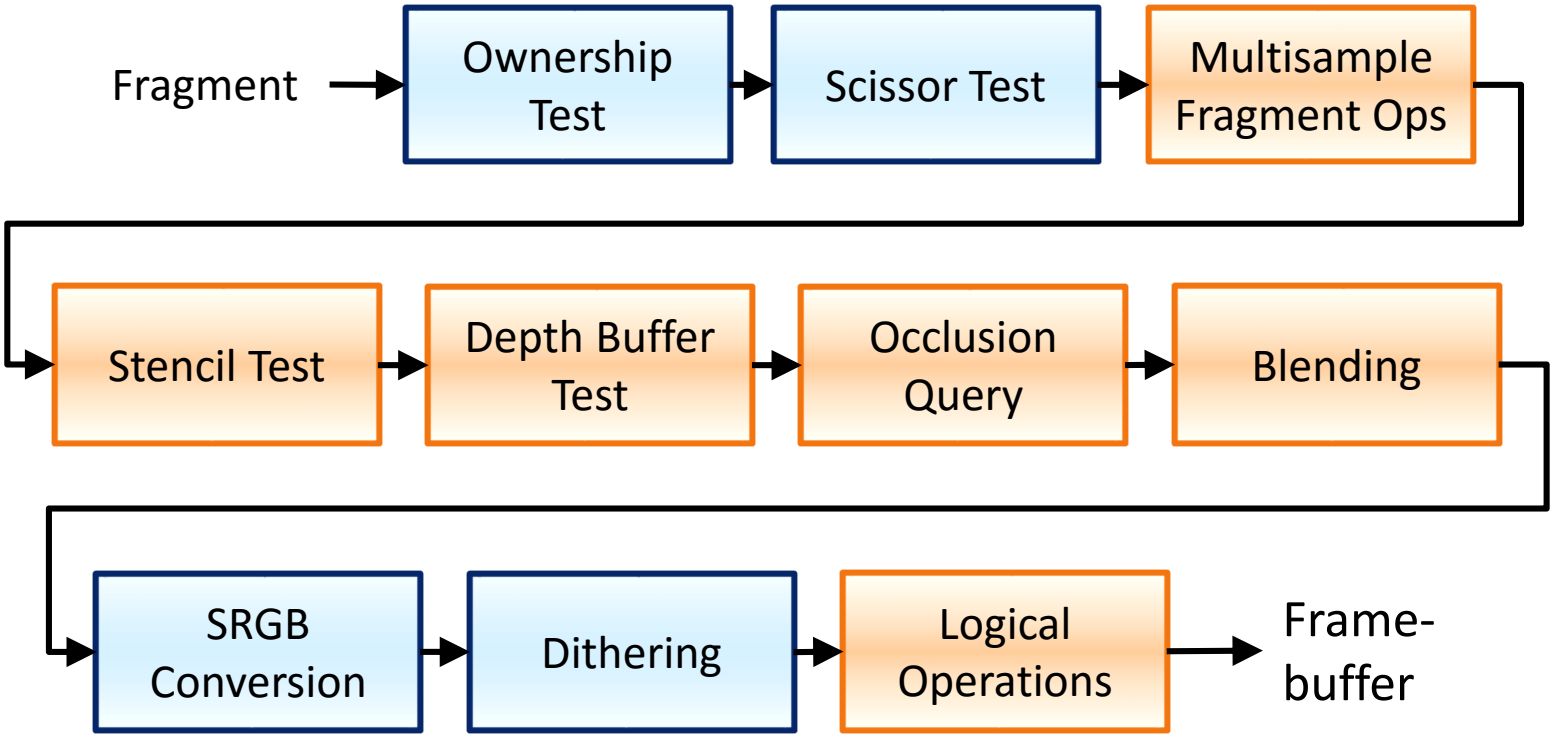
```
in vec3 world_position;
in vec3 world_normal_interpolated;

out vec4 frag_color;

void main() {
  vec3 world_normal = normalize(
    world_normal_interpolated);

  vec3 color = world_normal;
  color += vec3(1.f, 1.f, 1.f));
  color *= 0.5f;

  frag_color = vec4(color, 1.f);
}
```

- Gibt die Normaleninformation als Farbe aus
  - hilfreich zum Debuggen

Fragmentverarbeitung



- Fragmentoperationen sind sehr wichtig für Rasterisierungsverfahren
  - Tiefentest, Blending, …



Alpha Blending

Source Color (Fragmentfarbe)

Destination Color (Framebuffer bevor)

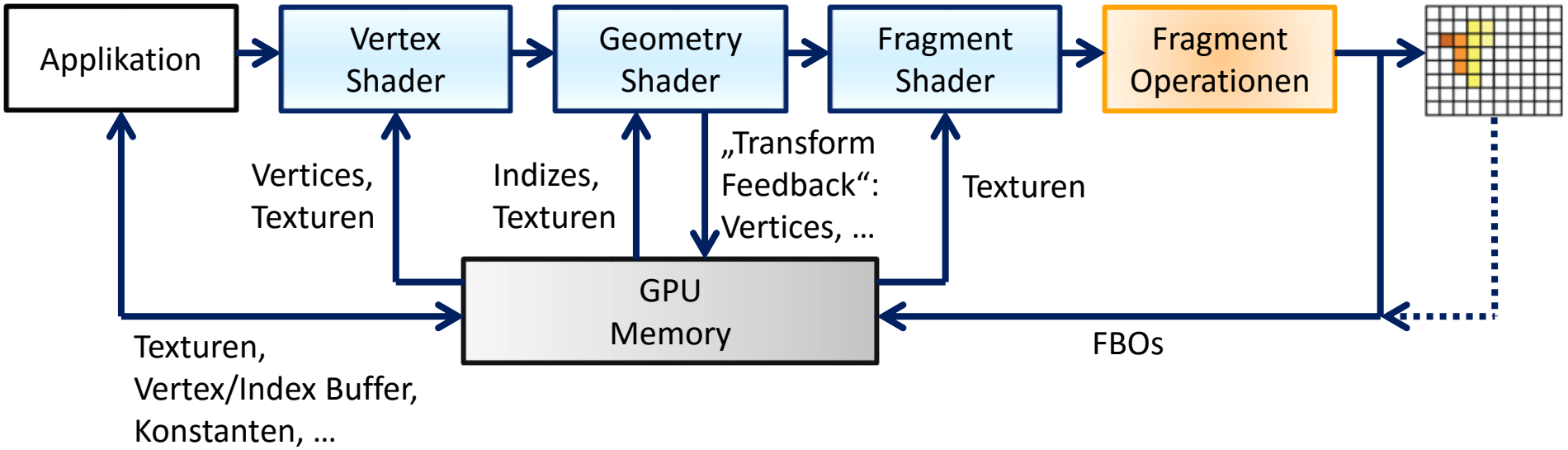
Source Alpha (Fragment Alpha)

Final Color (Framebuffer danach)

Final Color = Source Alpha * Source Color + (1 – Source Alpha) * Destination Color

Moderne OpenGL Pipeline

- eine „funktionale“ Sicht auf die zunächst wichtigen programmierbaren Teile der Pipeline



- u.a. hier nicht gezeigt:
  - jeder Shader kann lesend/schreibend auf Puffer zugreifen (Ausnahme: der Framebuffer kann nur geschrieben werden)

Programmierbare Pipeline bietet viel Flexibilität

Programmierbare Pipeline bietet viel Flexibilität

The Witness

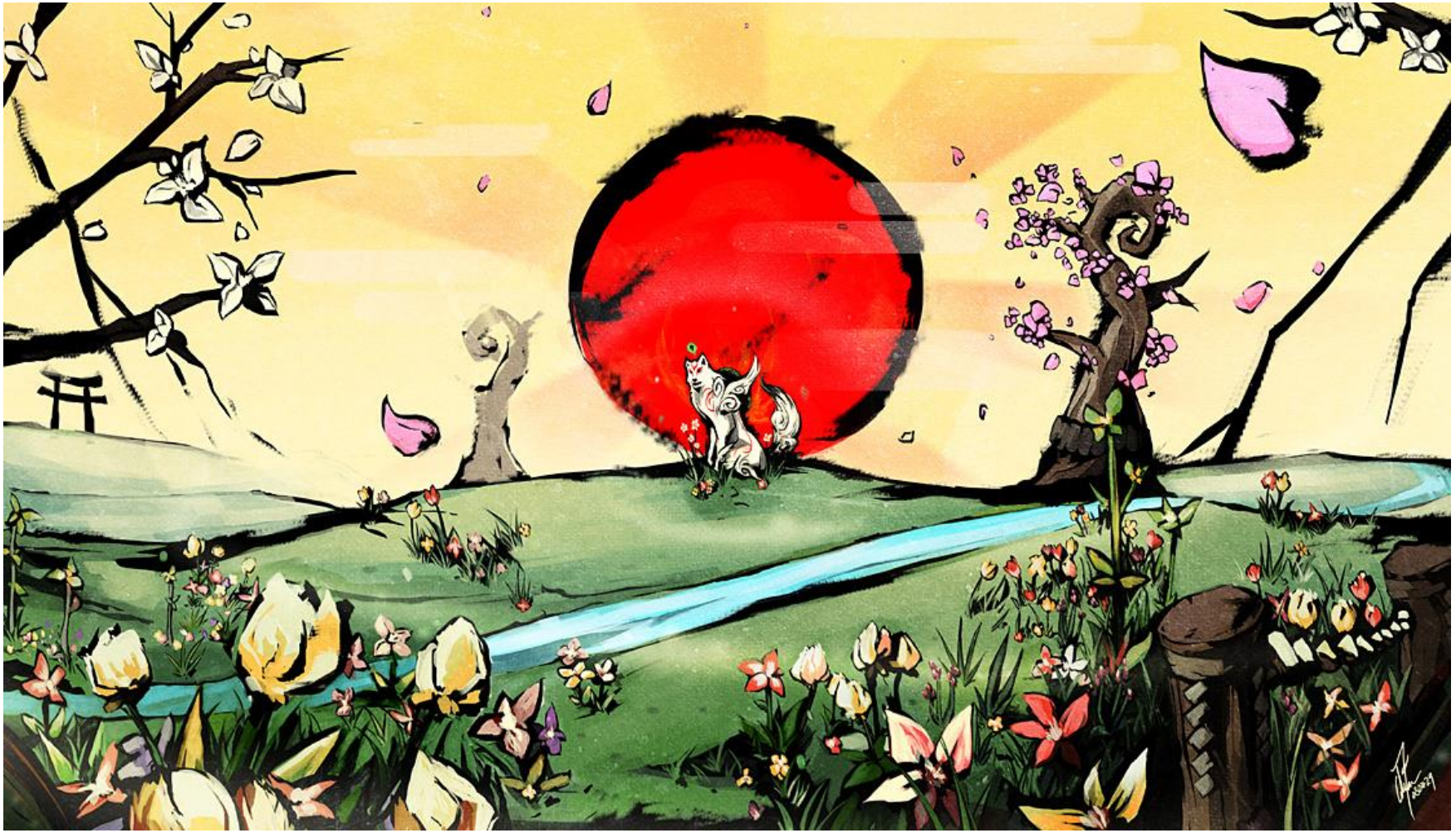
Programmierbare Pipeline bietet viel Flexibilität

Black: The Fall

Programmierbare Pipeline bietet viel Flexibilität

Rime

Programmierbare Pipeline bietet viel Flexibilität

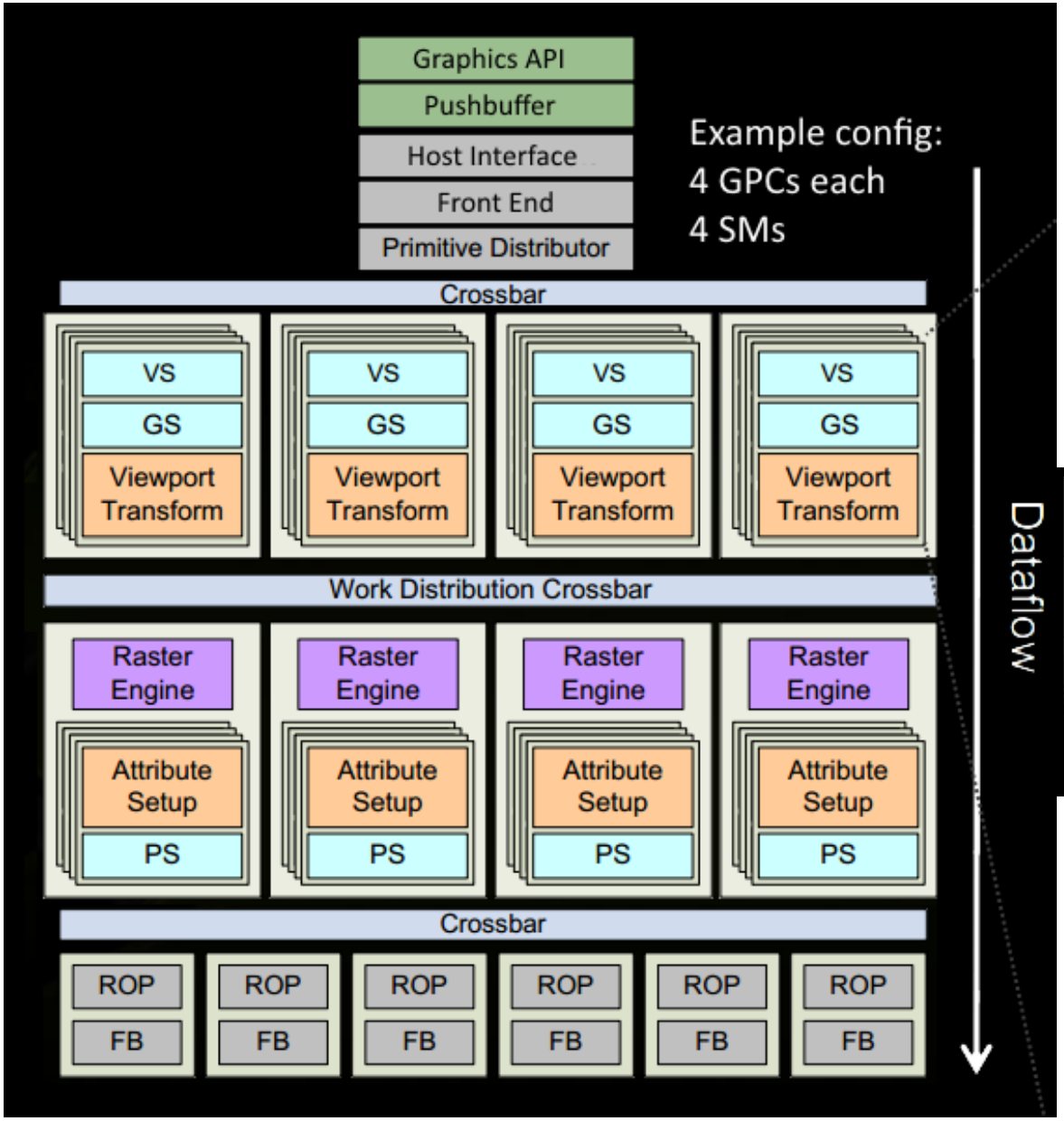


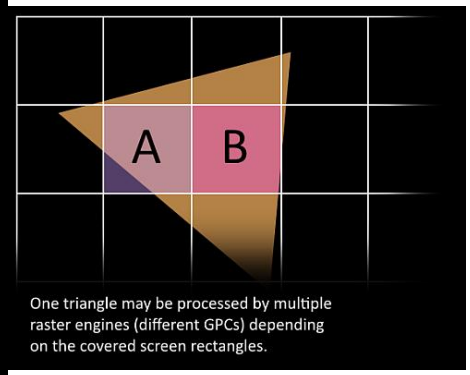
Okami (mehr Beispiele: http://www.themasquera.de/games/work-of-art/)

GPU Architecture: Case Study “Maxwell”

https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

GPU Architecture: Case Study “Maxwell”



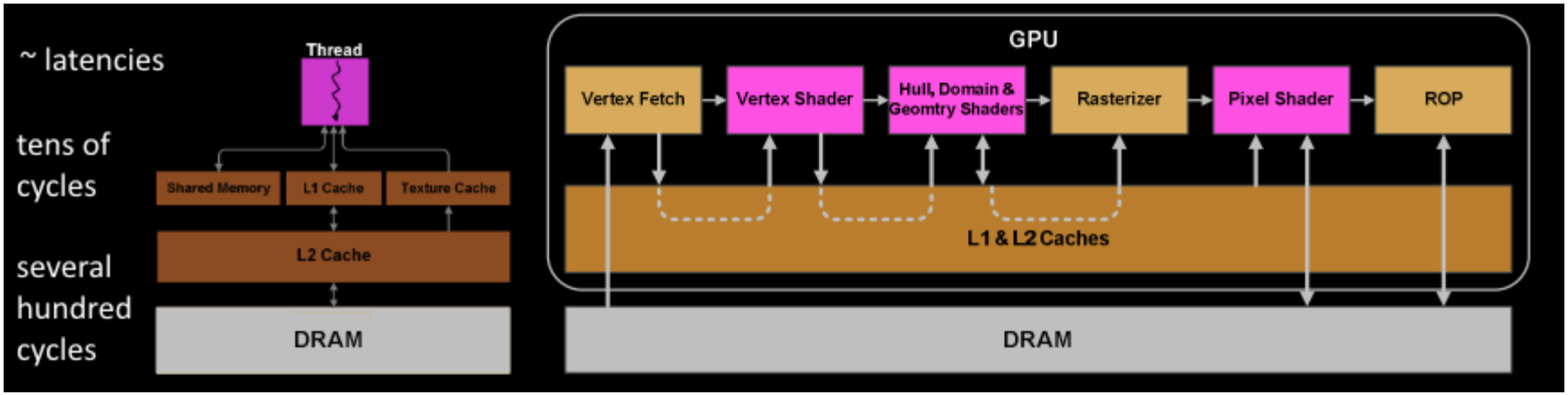


https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

GPU Architecture: Case Study “Maxwell”

- Speicherhierarchie
  - L1 Cache, Texture Cache: ~24 KB pro SM, schnell
  - L2 Cache: ~2 MB geteilt von allen SMs und Clusters, langsamer
  - Global (& Texture) Memory: ~4 GB, “schneller” DRAM, ~200 GB/s

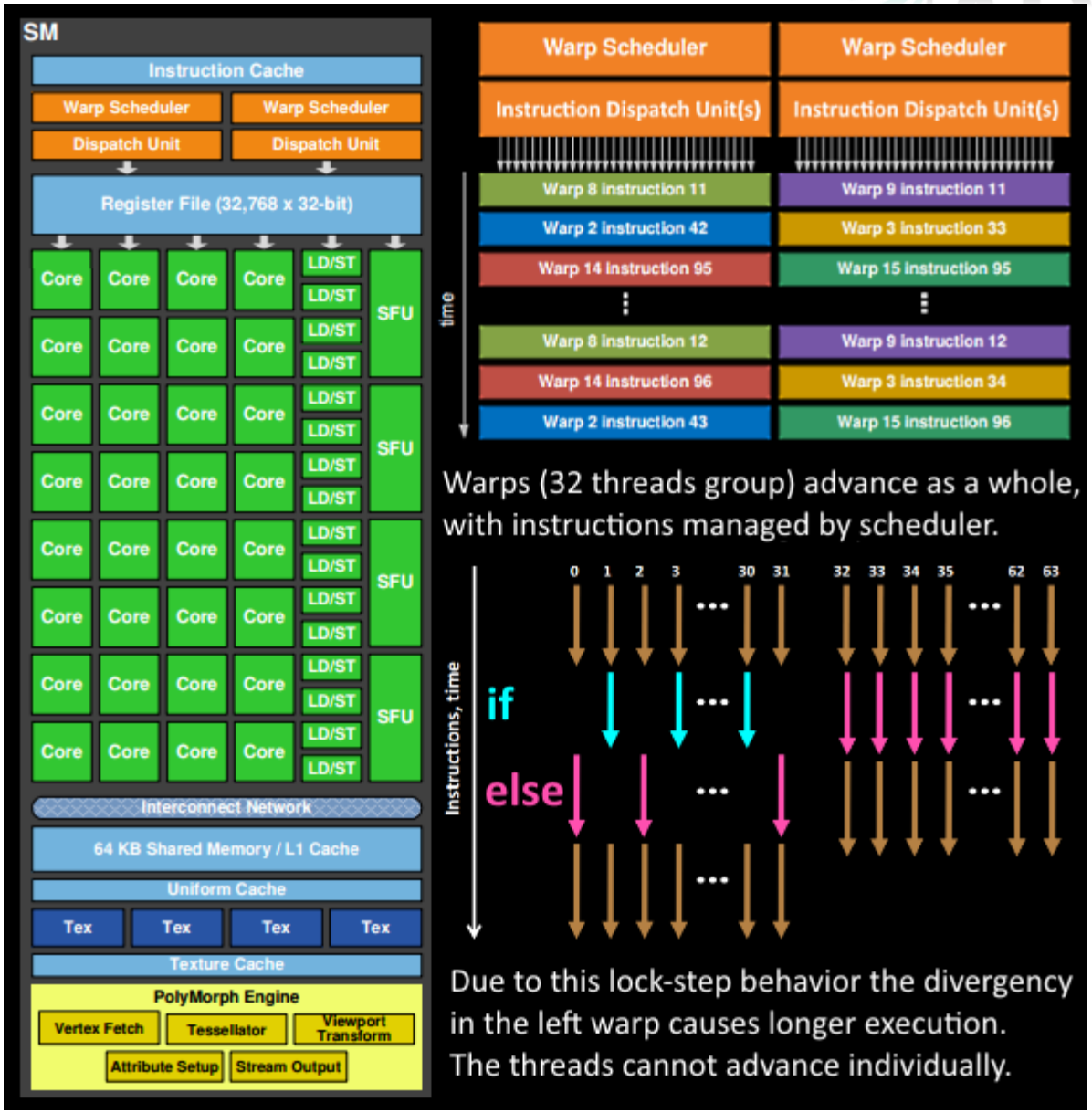
- Shared Memory:
  - nur lokal zugänglich
  - 96 KB pro SM, schnell



https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

GPU Architecture: Case Study “Maxwell”

- Viele 1000 Threads bereit zur Ausführung
- Latency Hiding durch Context Switch
  - großes Register File
  - Wenig Overhead
- Massiv parallele Aufgaben
  - Lockstep/SIMT: SIMD + Masking
  - Zusammengefasste Speicherzugriffe



https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

Fixed-function Hardware Today?

- Case Study hardward-accelerated Ray Tracing (PowerVR)

- **SCHEDULING / BANDBREITE / CACHE-LOKALITÄT!**
- Fixed-function Operations
  - Ray / AABB
  - Ray / Triangle

www.gdcvault.com/play/1020741/New-Techniques-Made-Possible-by