Architektur und Programmierung von Grafik- und Koprozessoren

Die Grafik Pipeline

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

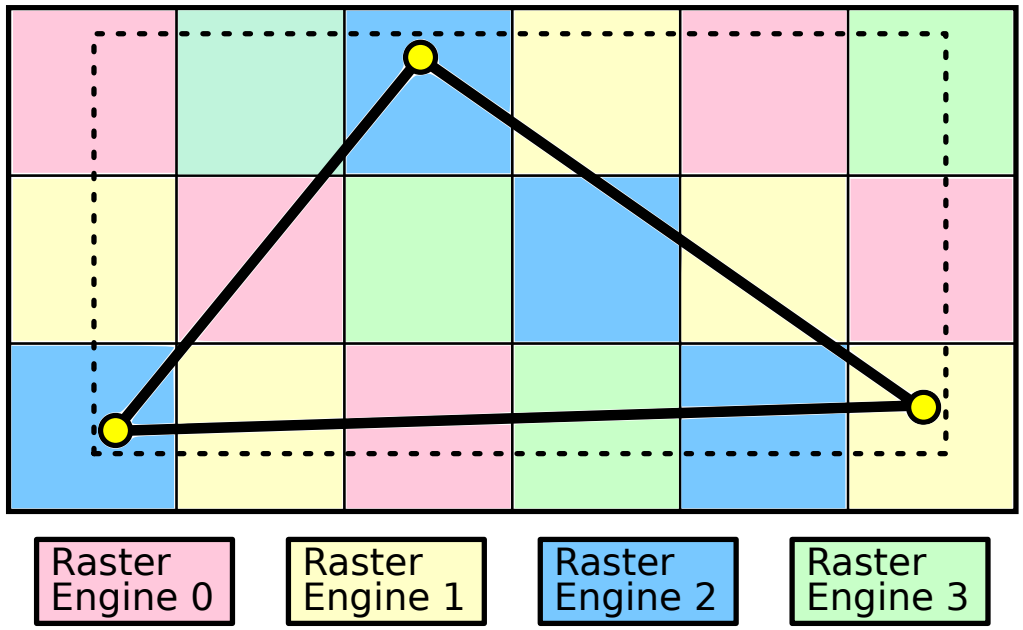
Scan Konvertierung

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

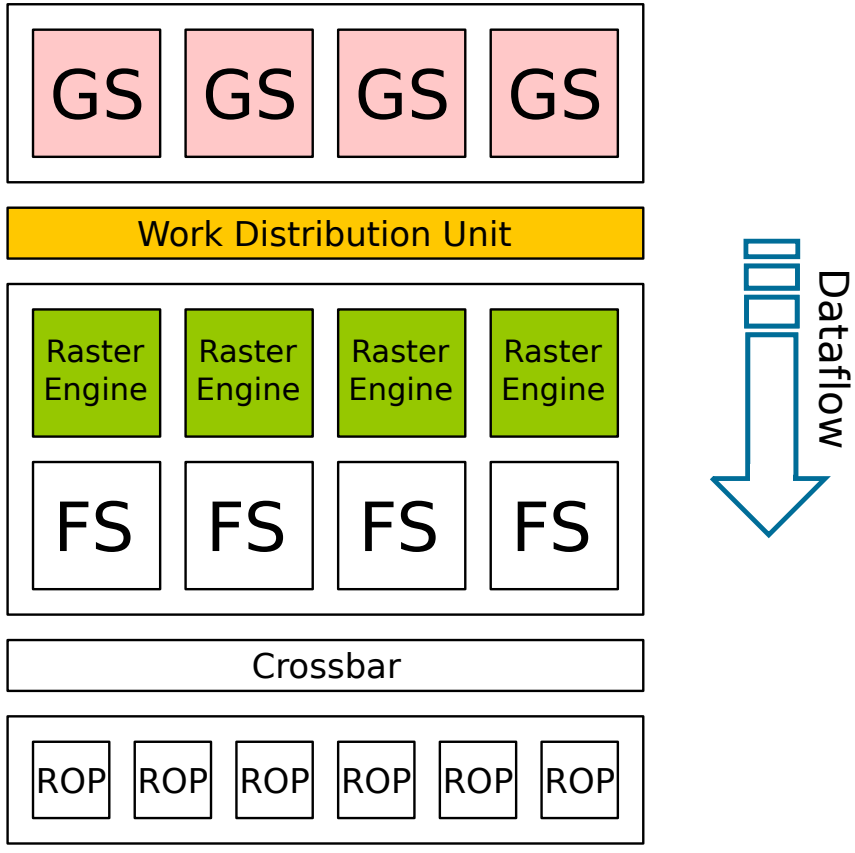
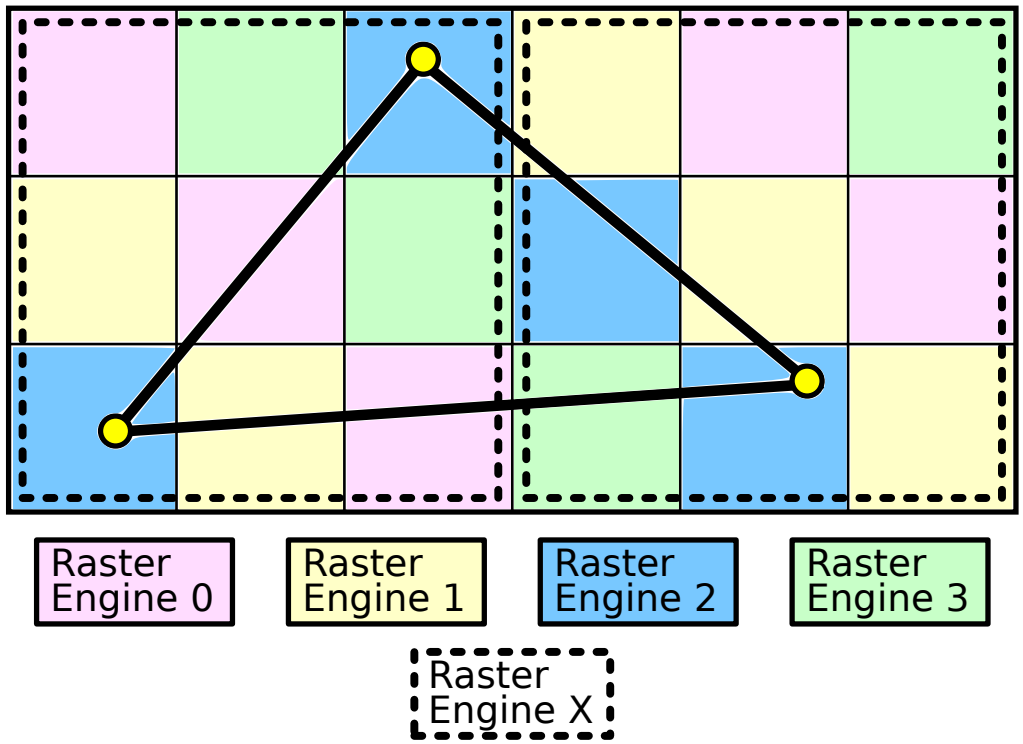


Abbildung: vereinfachend gemäß Tim Purcell: Fast Tessellated Rendering on Fermi GF100, HPG 2010. (GS = “Geometry Stage”, FS = “Fragment Stage”.)

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

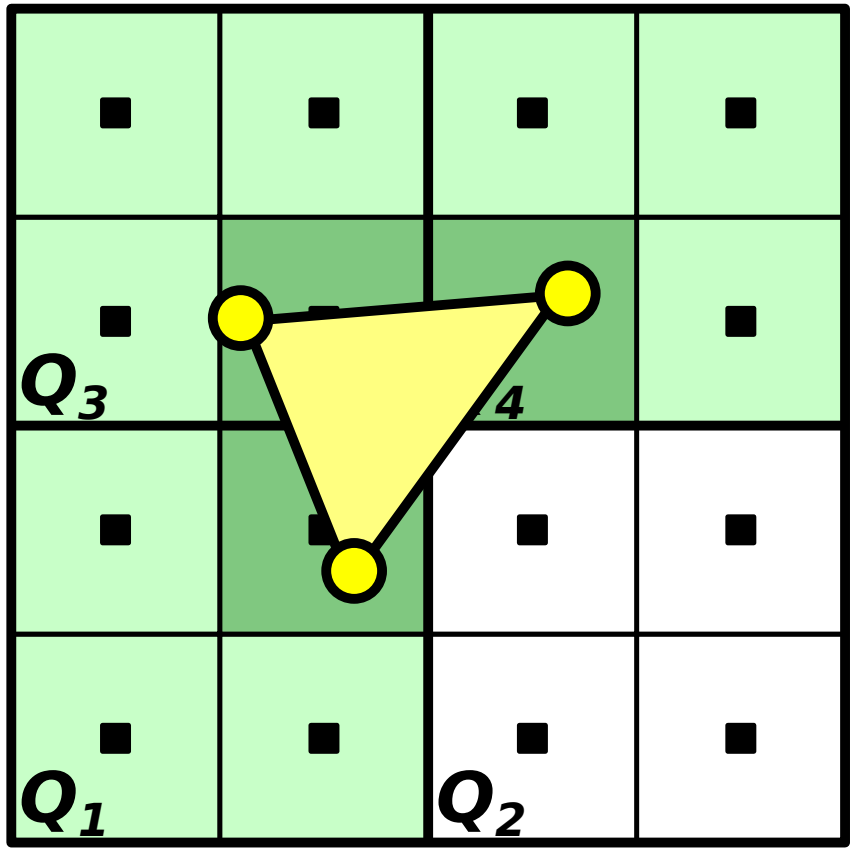


Abbildung: in Anlehung an Fabian Giesen: A trip through the graphics pipeline. Dreieck überdeckt nur drei Fragmente. Dennoch werden drei Quads mit insgesamt zwölf Einzelfragmenten erzeugt.

Scan Konvertierung

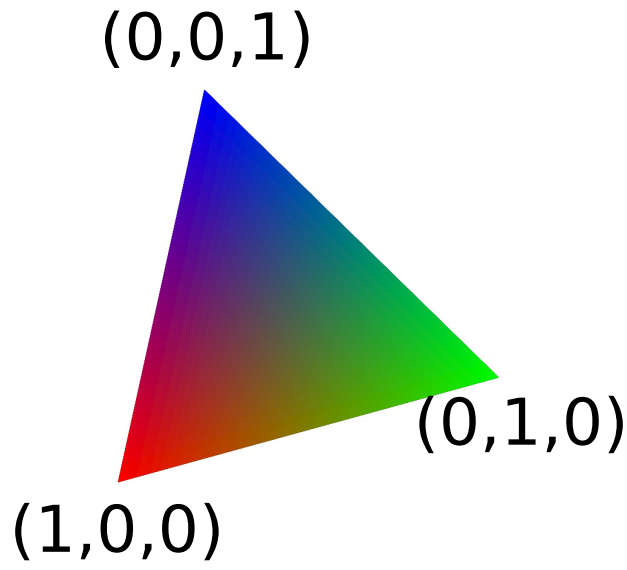
Triangle Setup

- 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.

Fragment Stage

Fragment Phase

- Während dieser Phase werden Fragmente texturiert und beleuchtet.
- Fragment Phase ist programmierbar (Fragment Programm).
- So wie Vertices werden Fragmente von Shader Cores verarbeitet.

Verteile Arbeit auf Shader Cores

- Raster Engines verarbeiten Kacheln, generieren Quads.
- Quads (2×2 Pixel) kommen von zwei, vier, ... Raster Engines und werden erst gepuffert (API Order).
- 32,64 o. ä. viele Threads in Thread Groups (Nvidia: Warps) bearbeiten jeweils ein Quad.
  - Unified Shader Architekturen $\Rightarrow$ Prozessoren und Scheduling Logik wie bei Vertex Stage.
- Bemerkung: Quads GPU "Elementarteilchen". In Fragment Programm stehen dedizierte Instruktionen für Derivatives zur Verfügung.

API Order und Sort-Last Fragment Shading

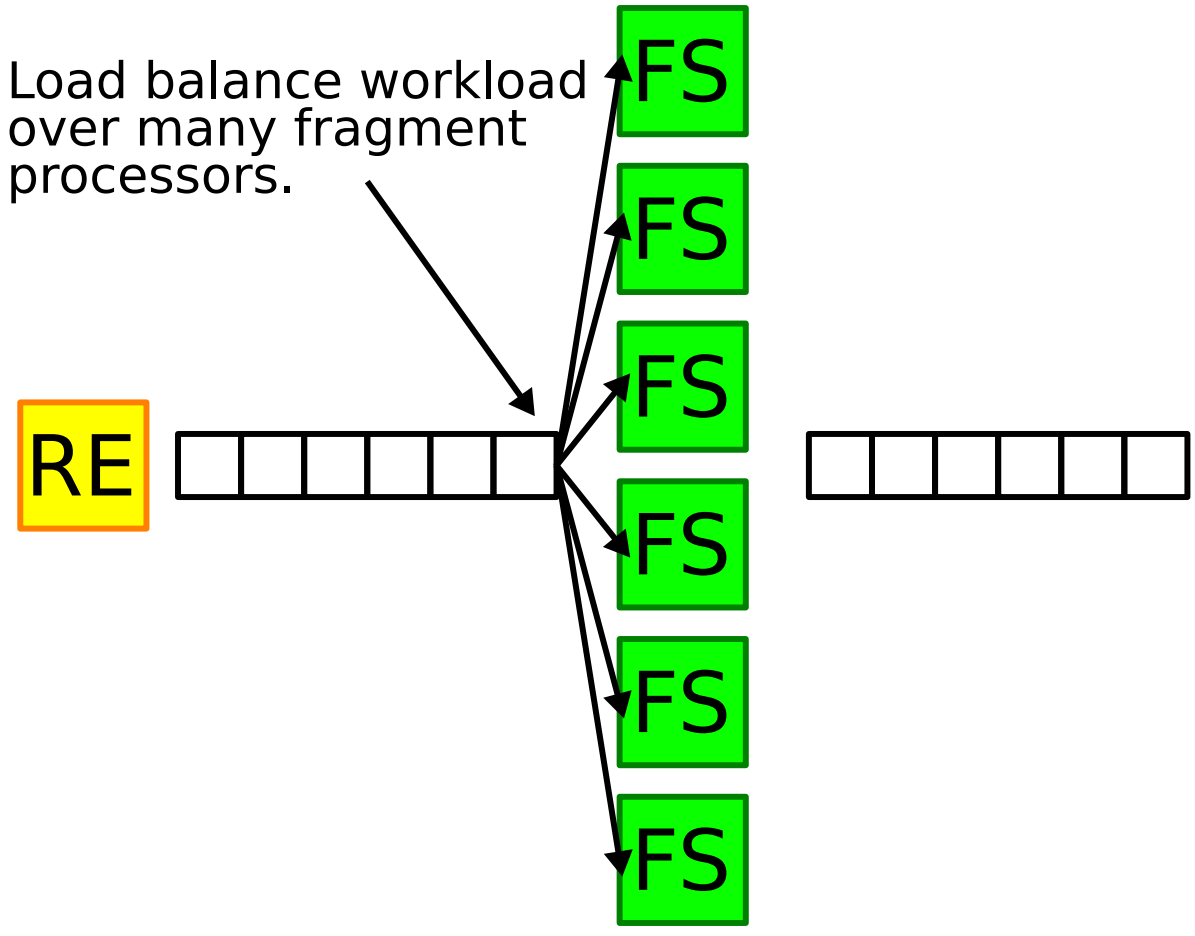


Abbildung: vgl. 1 Beyond Programmable Shading GPU architecture II: Scheduling the graphics pipeline, Mike Houston (AMD) Aaron Lefohn (Intel).

API Order und Sort-Last Fragment Shading

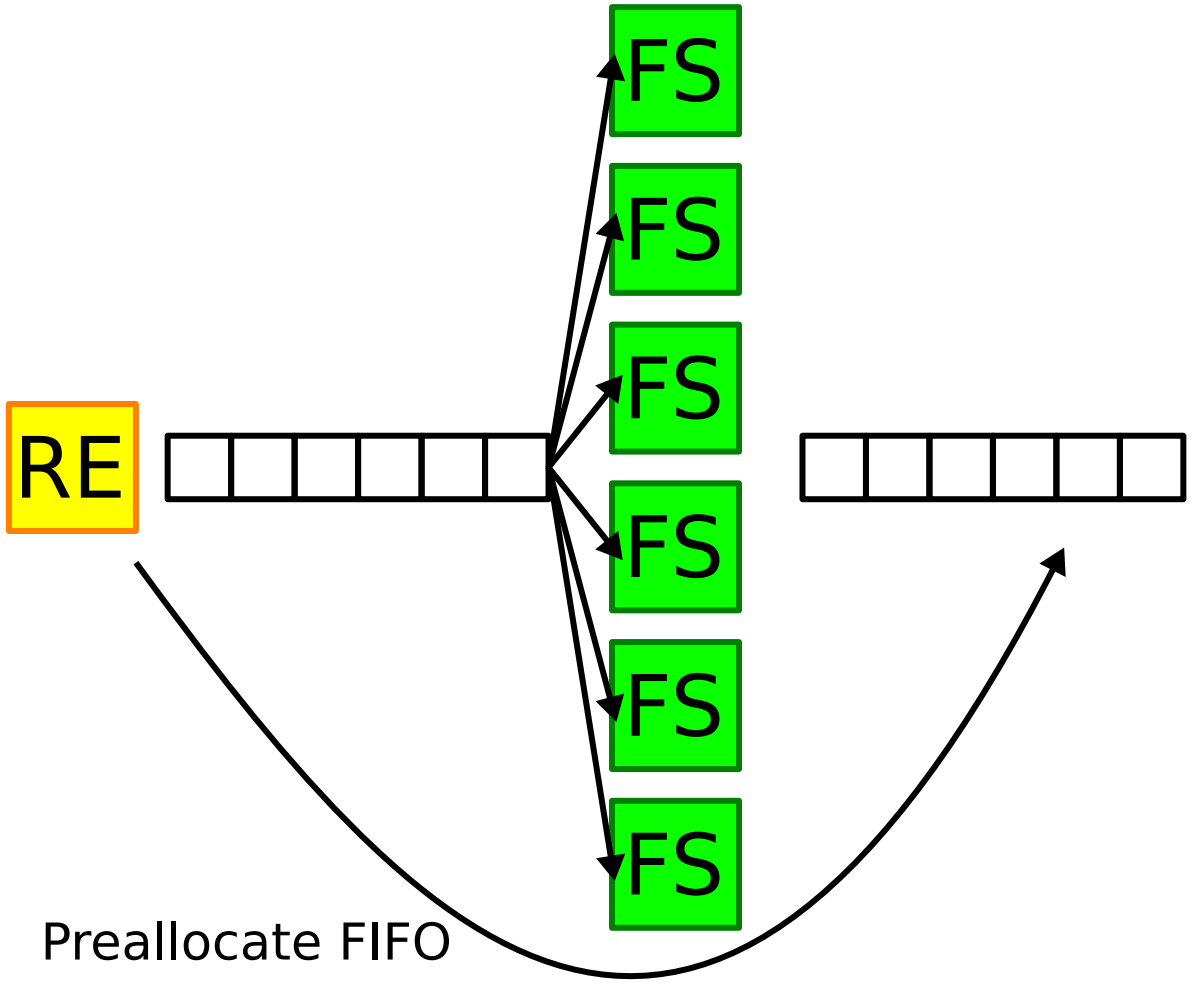


Abbildung: vgl. 1 Beyond Programmable Shading GPU architecture II: Scheduling the graphics pipeline, Mike Houston (AMD) Aaron Lefohn (Intel).

API Order und Sort-Last Fragment Shading

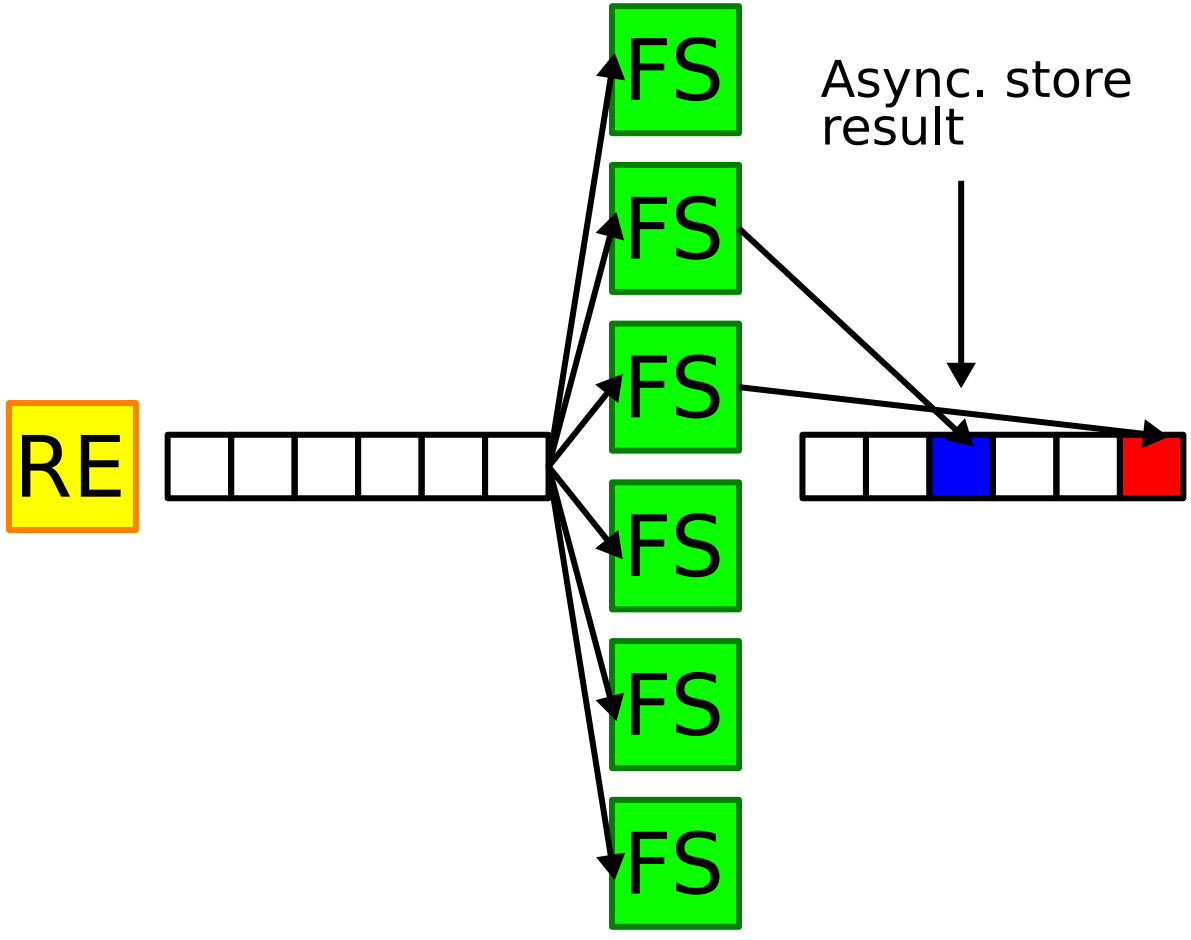


Abbildung: vgl. 1 Beyond Programmable Shading GPU architecture II: Scheduling the graphics pipeline, Mike Houston (AMD) Aaron Lefohn (Intel).

Tiefentest und Alpha Blending

Framebuffer Operationen

Letzte Phase in Grafik Pipeline: Compositing Operationen (Tiefe, Alpha); Antialiasing; schreibe Ergebnis in Framebuffer.

- Framebuffer: i. d. R. dedizierter Speicherbereich für die finalen Bilddaten (Farbe, ggf. Tiefe). *Display Prozessoren* bedienen sich aus diesem Speicher.
- Meistens mit Double Buffering Logik implementiert.
- GPUs unterstützen auch "offscreen rendering": *Framebuffer Objects* (FBOs). Werden mit Texturen für Farbe und ggf. Tiefe ("Color Attachment" und "Depth Attachment") initialisiert. Texturen können später in anderem Kontext gebunden werden.

Render Output Einheiten (ROPs)

- ROPs: dedizierte fixed-function Hardware, die Compositing Operationen implementiert.
- ROPs nicht frei programmierbar, jedoch parallele Prozessoren.
- z. B. Nvidia Maxwell Architektur: 64 ROPs.
- Spätestens hier muss API Order hergestellt sein.

Framebuffer Kompression

- Kompression, um effektive Bandbreite bei Framebuffer Fetches zu erhöhen (nicht um Speicher zu sparen).
- APIs fordern *verlustfreie* Kompression.
- vgl.: Nvidia GTX 980 Whitepaper: GM204 Hardware Architecture In-Depth
- Bilddaten werden kachelweise in DDR Framebuffer geschrieben.
- Prinzip von Kompressionsmethoden ähnlich *run-length encoding*.

Framebuffer Kompression

Run-length encoding Beispiel

a.)

Uncompressed sequence: 8b

0	0	0	0	0	1	1	1

Compressed sequence: 4b

5x0	3x1

b.)

Uncompressed sequence: 8b

0	1	2	3	4	5	6	7

"Compressed" sequence: 16b

1x0	1x1	1x2	1x3
1x4	1x5	1x6	1x7

Framebuffer Kompression

Delta Color Compression

Idee: verarbeite Blöcke von Pixeln. Ist die Farbe zweier Pixel sehr ähnlich, speichere Farbe $1\times$, speichere außerdem die Differenz zwischen den beiden Pixeln mit viel weniger Bits.

Problem: Verfahren müssen verlustfrei sein. Daher implementieren GPUs Heuristiken, z. B.: lässt sich eine Kachel von 4×2 Pixeln delta-komprimieren? Falls nein, teile in zwei 2×2 Kacheln auf, falls auch so keine Kompression möglich, komprimiere gar nicht.

Moderne GPUs sparen signifikant viel Bandbreite Deltakompression, vgl. Vortrag von Nvidia: *GTX 1080 Pascal Memory Compression Explained* `https://www.youtube.com/watch?v=yblANXXMuds` (abgerufen am 30.5.2018).

Framebuffer Kompression

Delta Color Compression Beispiel

0.5	0.6	0.7	0.8	0.9	1.0
0.5	0.6	0.7	0.8	0.9	1.0
0.5	0.6	0.7	0.8	0.9	1.0
0.5	0.6	0.7	0.8	0.9	1.0

0.5	0.1	0.1	0.1	0.1	0.1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

- Verfahren besonders gut für Farbverläufe geeignet.
- (Verläufe entstehen z. B. beim Interpolieren.)

Framebuffer Kompression

Huffman Coding

- Verlustloses Transkodieren von Nachrichten (Text, Bilder,etc.).
- Transkodiere so viel Information wie möglich mit so wenig Bits wie möglich.
- Teil von Deflate (kommt z. B. bei zip, gzip und png zum Einsatz).
- Kodiere Zeichen (ASCII, Farbintensität,etc.) mit variabler Länge, häufig vorkommende Zeichen $\Rightarrow$ weniger Bits.
- Nachteil: wie bei RLE - komprimierte Nachricht kann ggf. länger sein als unkomprimierte Nachricht.

Framebuffer Kompression

Huffman Coding

Gegeben: Alphabet Σ, Wort w der Länge n über Σ.

Konstruiere Präfixbaum / Trie, sodass jedem Zeichen aus Σ ein eindeutiges Präfix zugewiesen wird.

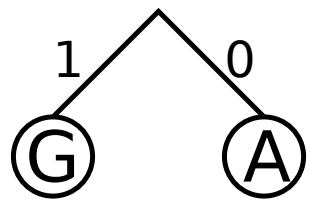
1. Erzeuge Liste mit Tupeln $(a, p(a))$, wobei $p(a)$ die Häufigkeit von a in w.
2. Führe je zwei Paare mit minimalem $p(a), p(b)$ in der Liste zu einem binären Teilbaum zusammen, sodass Tupel mit $max(p(a), p(b))$ (beliebig) linker Teilbaum. Label der linken und rechten Kante: 1 und 0.
3. Entferne die Tupel aus der Liste, ersetze durch Tupel $(\{a, b\}, p(a) + p(b)$ ($\Rightarrow$ neues Alphabet Σ').
4. Wiederhole, bis nur noch ein Tupel in der Liste.

Framebuffer Kompression

Huffman Coding Beispiel

Gegeben: $\Sigma = \{A, C, G, T\}$, $w = TTTTTTACCGCGTTCCCGTT$

$a \in \Sigma$	$p(a)$
A	1
C	6
G	3
T	10

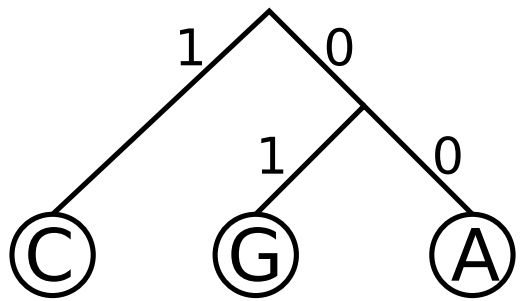


Framebuffer Kompression

Huffman Coding Beispiel

Gegeben: $\Sigma = \{A, C, G, T\}$, $w = TTTTTTACCGCGTTCCCGTT$

$a \in \Sigma'$	$p(a)$
C	6
T	10
GA	4

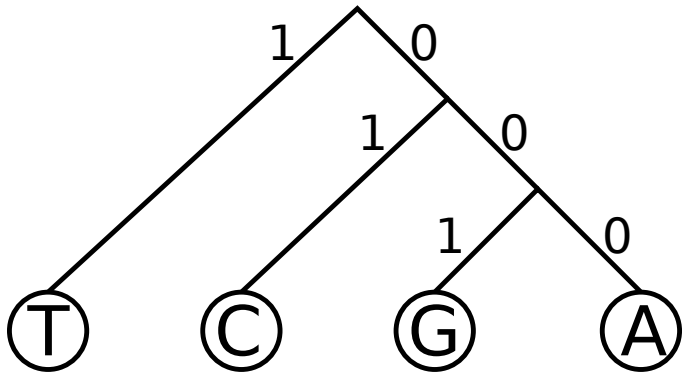


Framebuffer Kompression

Huffman Coding Beispiel

Gegeben: $\Sigma = \{A, C, G, T\}$, $w = TTTTTTACCGCGTTCCCGTT$

$a \in \Sigma''$	$p(a)$
T	10
CGA	10

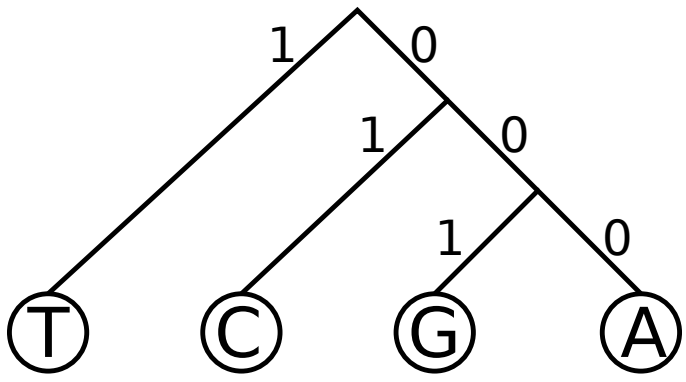


Framebuffer Kompression

Huffman Coding Beispiel

Gegeben: $\Sigma = \{A, C, G, T\}$, $w = TTTTTTACCGCGTTCCCGTT$

$a \in \Sigma'''$	$p(a)$
TCGA	20



Huffman Codes:

A = 000

C = 01

G = 001

T = 1

Framebuffer Kompression

Huffman Coding

Offensichtlich: Zeichen, die selten vorkommen, liegen tief im Baum und werden mit mehr Bits transkodiert, als Zeichen, die häufig vorkommen.

Voraussetzung: Eindeutigkeit - kein Huffman Code ist Präfix eines anderen Huffman Codes.

Eindeutigkeit folgt aus Präfixbaumeigenschaft. Informell: sei (a, b) Teilbaum mit Knoten a und b sowie Kantengewichten 1 und 0.
$\Rightarrow$ kein Blatt aus Teilbaum a kann sich Präfix mit Blatt aus Teilbaum b teilen. Da jeder Teilbaum des Huffman Baums diese Struktur hat, ist der Huffman Baum ein Präfixbaum und alle Blätter haben eindeutiges Präfix.

Texturkompression

S3 Texturkompression

- Anders als Render Output Compression: verlustbehaftet, explizites Texturformat, das per API eingeschaltet werden muss.
- Die meiste Grafik Hardware unterstützt komprimierte 2D Texturen.
  - Meistens S3 Texturkompression (S3TC).
  - OpenGL Extension `EXT_texture_compression_s3tc`
- Spezielle Formate, die schnelle *Dekompression* in Hardware erlauben.
- Varianten von S3TC: DTX1 und DTX3 (DX = **D**irect**X**).

S3 Texturkompression

Prinzip: bestimme zwei (möglichst repräsentative) Farben (Farbe-0 und Farbe-1) für *Bildschirmkacheln*, speichere Bitmap mit Codes, die beschreiben, wie zwischen den beiden Farben interpoliert wird.

Verlustbehaftet, Güte abhängig von Varianz im Bild, und von Entscheidung, welche Farben repräsentativ.

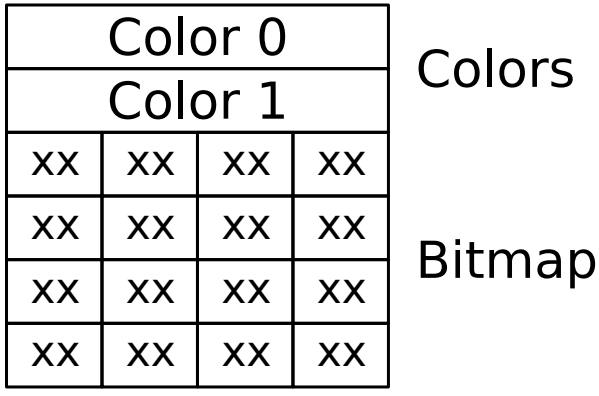


Abbildung: vgl. Ignacio Castãno: High Quality DXT Compression using CUDA (2007).

S3 Texturkompression

DTX1 Format

RGB Kompressionsformat mit Kompressionsrate 6:1 (ggü. 24-bit RGB).

- Textur unterteilt in 4×4 Blöcke.
- Speichere 64-bit pro Block:
  - $2\times$ 16-bit Farbe (`RGB_5_6_5`).
  - 32-bit Integer, je 2 bit Code assoziiert mit jedem Farbkanal eines Pixels in Kachel.
  - Code 0 bzw. 1: Pixel := Farbe-0 bzw. Farbe-1.
  - Code 2: Falls Farbe-0 $>$ Farbe-1: $(2 \times \text{Farbe-0} + \text{Farbe-1})/3$, sonst $(\text{Farbe-0} + \text{Farbe-1})/2$.
  - Code 3: Falls Farbe-0 $>$ Farbe-1: $(\text{Farbe-0} + 2 \times \text{Farbe-1})/3$, sonst 0.

S3 Texturkompression

DTX3 Format

RGB*A* Texturkompressionsformat, 128-bit pro Kachel, Kompressionsrate $4:1$ im Vergleich zu 32-bit RGBA.

Zwei 64-bit "Daten-Chunks". Erstes Chunk speichert 16×4 Alpha Werte, zweites Chunk bis auf kleine Berechungsunterschiede analog zu DTX1.

S3 Texturkompression

- Dekompression äußerst einfach zu implementieren, insb. in Hardware.
- Dafür Kompression schwierig. Finden von repräsentativen Farben und Indices ein Optimierungsproblem.
- Wird i. d. R. auf lineares Optimierungsproblem abgebildet -
  1. Bestimme eine Anordnung von Indices in Bitmap.
  2. Die *Farbpalette* für die unkomprimierte Palette ist bekannt. Bestimme bei gegebenen Indices die zwei Farben so, dass die Distanz zu den Farben in Farbpalette minimal.
  3. Wiederhole für weitere Anordnungen von Indices.
- Für Welche Anordnungen von Indices das Verfahren wiederholt wird, hängt von Heuristik ab.