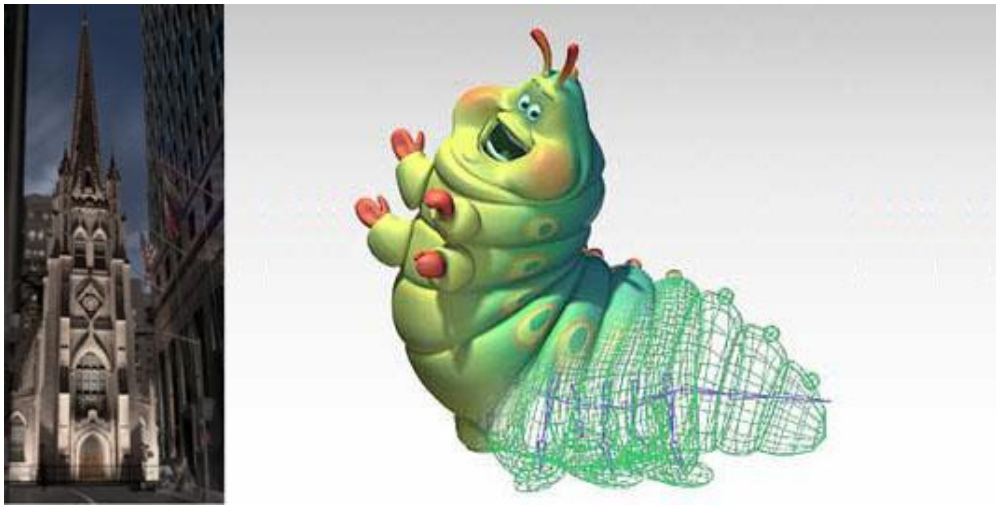
Übersicht

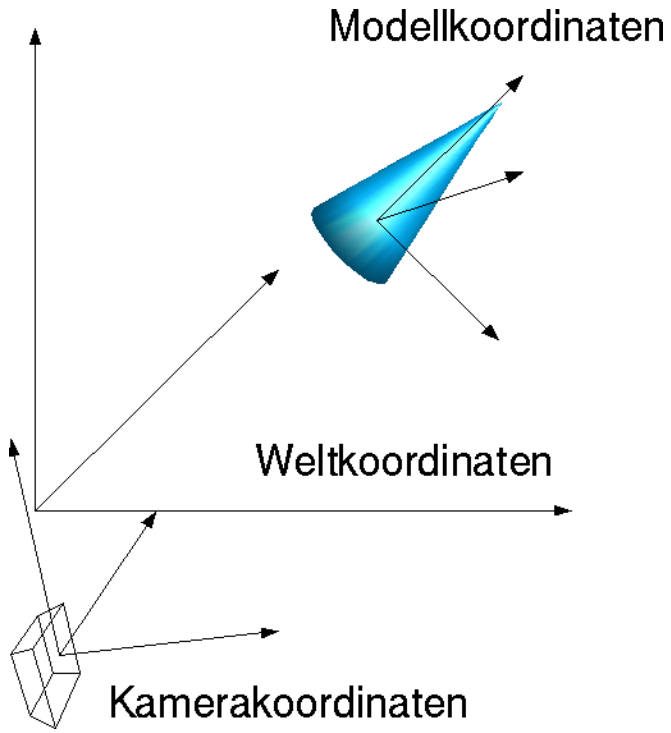
1. Grundlagen 3D Computergrafik
2. Polygongrafik, Z-Buffer
3. Texture-Mapping/Shading
4. GPU
5. Programmierbare Shader
6. GPGPU

3D Computergrafik

- 3D Transformationen
- 3D/2D Projektion
- Schattierung
- Texturen
- Zeichnen von Polygonen mit Tiefeninformation



3D Transformation



Unterscheidung zwischen:

- Weltkoordinatensystem
- Kamerakoordinatensystem
- Modellkoordinatensystem

3D Transformation

- Benutzung von *homogenen* Koordinaten

  (vgl. Projektive Geometrie)

⇨ Ergänzung um homogene Komponente

Bspw.: Vektor $\mathbf{p} \in \Re^3$ ⇨ $\begin{bmatrix} p_x \\ p_y \\ p_z \\ p_w \end{bmatrix}$

- Kompakte Darstellung von Translationen/Rotationen
- Projektion durch homogene Division

3D Transformation

Beispiel Koordinatentransformation:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Rotation/Skalierung

Translation

Beispiel Perspektivische Projektion:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

$$\begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x_s & y_s & z_s \end{bmatrix}$$

(homogene Division)

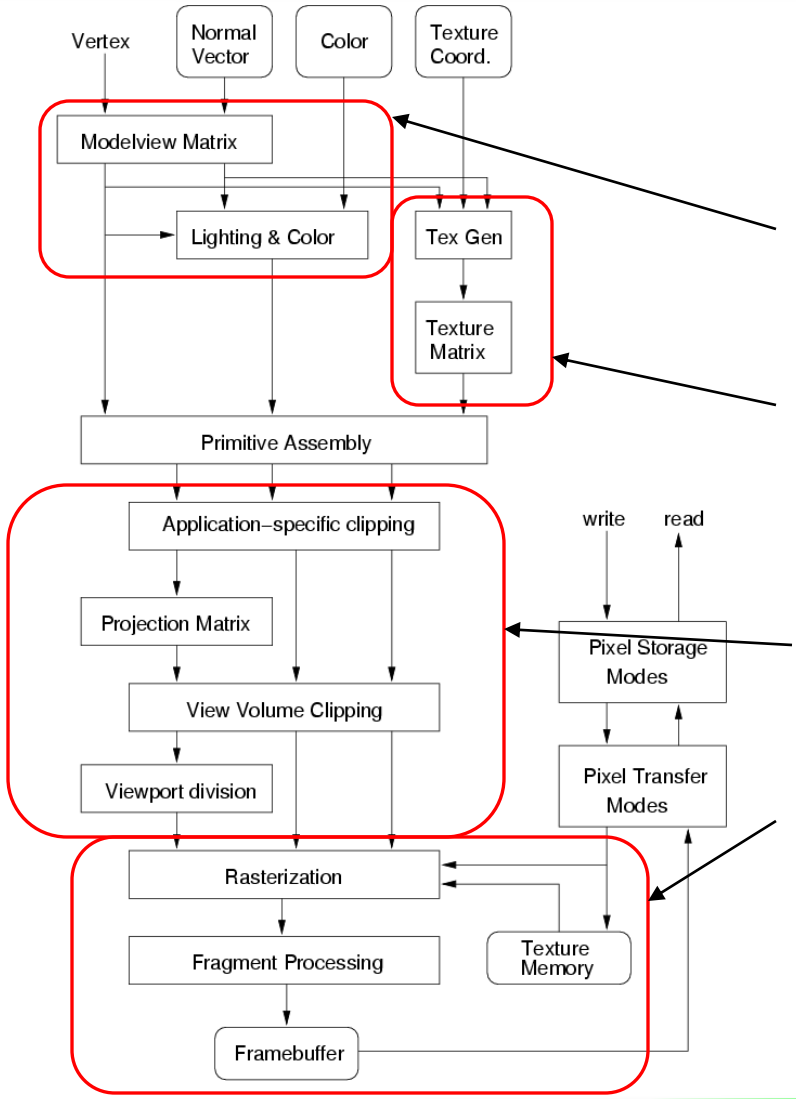
3D Beschleunigung – Antreibende Faktoren

- Gestiegene Anforderungen im CAD Bereich
- Simulatoren (speziell mil. Anwendungen)
- Visualisierung großer Datenmengen

3D Beschleunigung

Computerspiele

3D Grafikpipeline (OpenGL)



- Transformation & Beleuchtung
- Texturkoordinaten (Erzeugung und Interpolation)
- Clipping und Projektion
- Verrasterung

Zeichnen von Polygonen

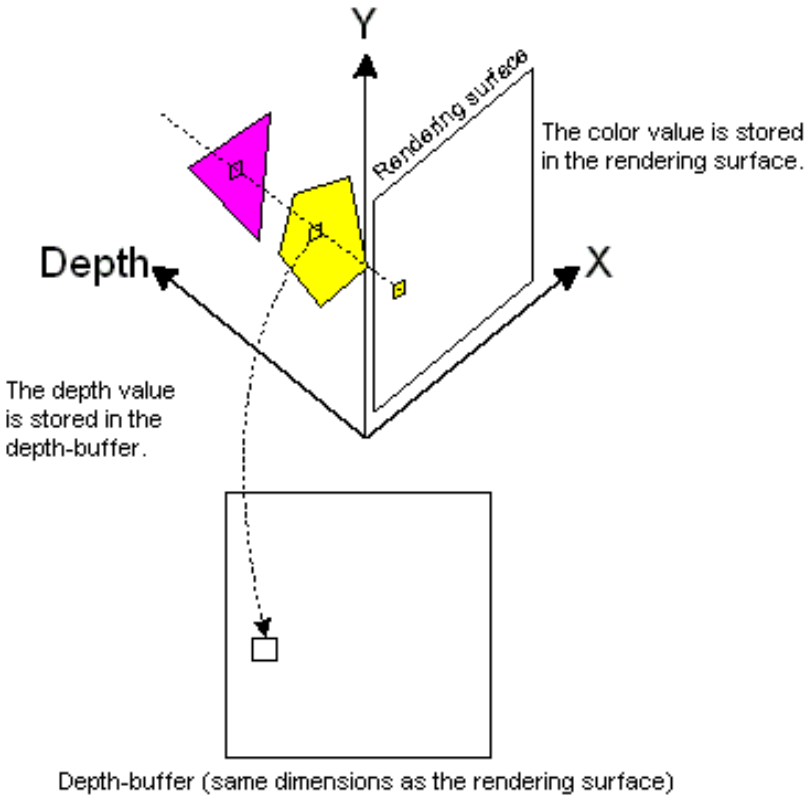
- 1. Generation der 3D-Beschleunigung
- CPU stellt transformierte Daten (ggf. Texturen, Farben)
- Zeichnen von Polygonen durch Grafikhardware
- Interpolation von Farbe, Texturkoordinaten, Z-Koordinaten

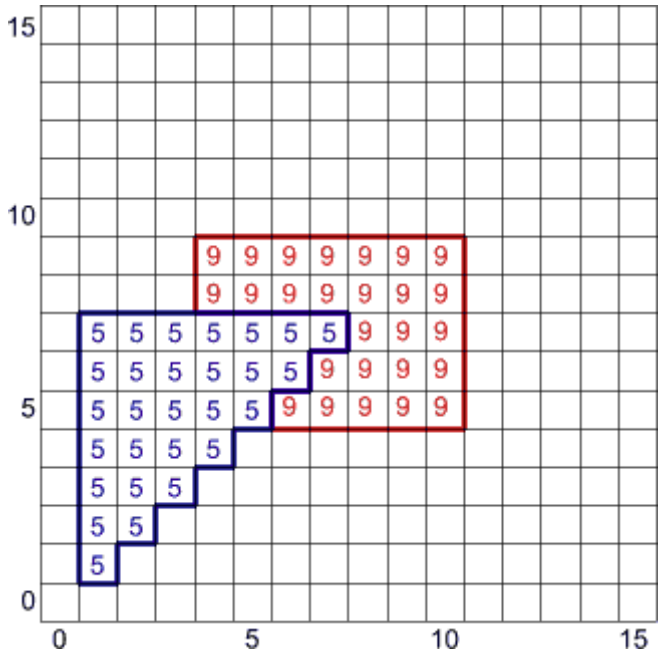
Problematisch:

- Verdeckungen aufgrund der 3D Daten

⇨ ***Z-Buffer***

Z-Buffer





- Speicher auf Grafikkarte (üblicherweise im Framebuffer)
- Speichert *n*-Bit Tiefeninformationen pro Pixel
- Hardware führt Vergleiche während des Zeichnens aus

Texture-Mapping

- Hinzufügen von Detailinformationen auf Polygone
- Detailinformationen sind i.A. 2D-Bilddaten
- Erhöhung des Fotorealismus
- Rechenaufwendig

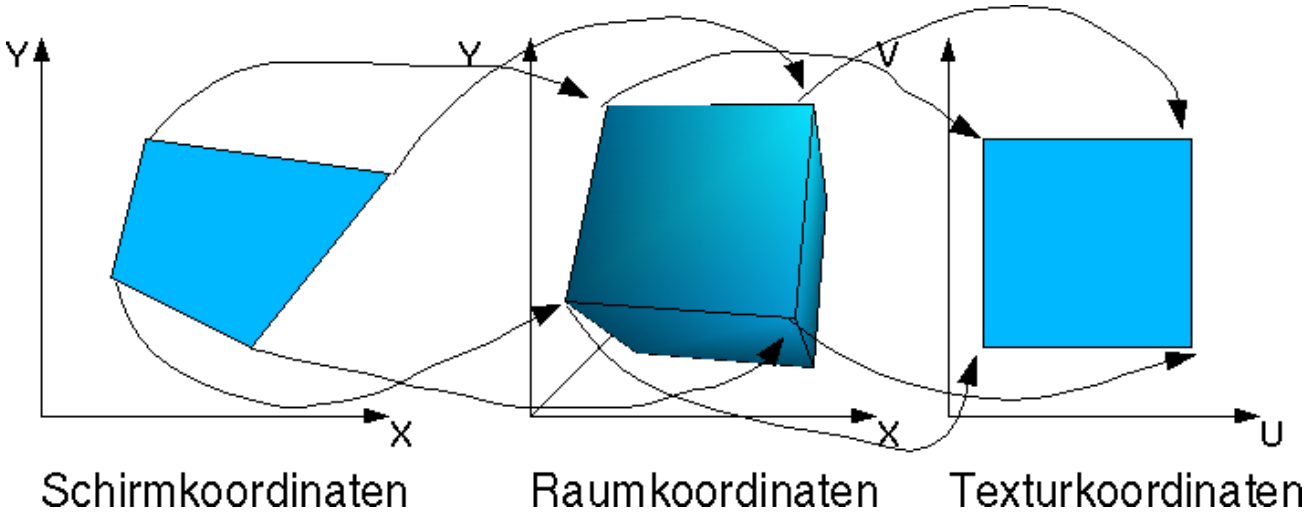
Eingabevektor für die Grafikhardware:

X/Y/Z/W Koordinatenvektoren, Texturkoordinaten (*u,v*), Textur-ID

⇨ Texturspeicher / ALU auf der Grafikkarte

Texture-Mapping

- Üblicherweise: Mapping Schirmkoordinaten -> Raumkoordinaten



Problem:

- Rückprojektion für jeden Bildschirmpixel

⇨ *Rational-Lineare Interpolation*

Texturen – RL Interpolation

- Projektion/Homogene Division an den Eckpunkten
- Lineare Interpolation von $\left[\frac{x}{w} \quad \frac{y}{w} \quad \frac{z}{w} \quad \frac{1}{w} \quad \frac{u}{w} \quad \frac{v}{w}\right]$
- Für jeden Pixel:

$$u = \frac{u/w}{1/w} \qquad v = \frac{v/w}{1/w}$$

⇨ Gute Realisierbarkeit in Hardware

Shading

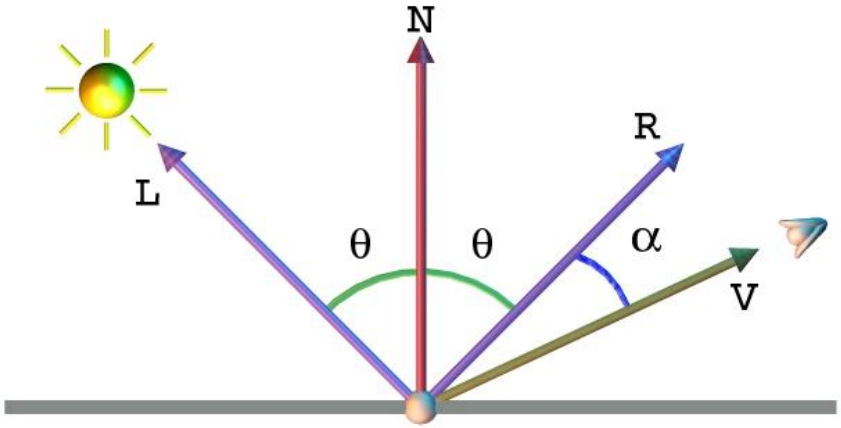
- Einbeziehen von Lichtquellen
- Erhöhung des Fotorealismus

- Üblicherweise lokale Beleuchtungsverfahren
  - Flat Shading
  - Gouraud Shading
  - Phong Shading

Shading

$$I = I_a k_a + f_{att} I_l (k_d (\mathbf{N}^{\mathrm{T}} \mathbf{L}) + k_s (\mathbf{R}^{\mathrm{T}} \mathbf{V})^n)$$

(Phong Beleuchtungsmodell)

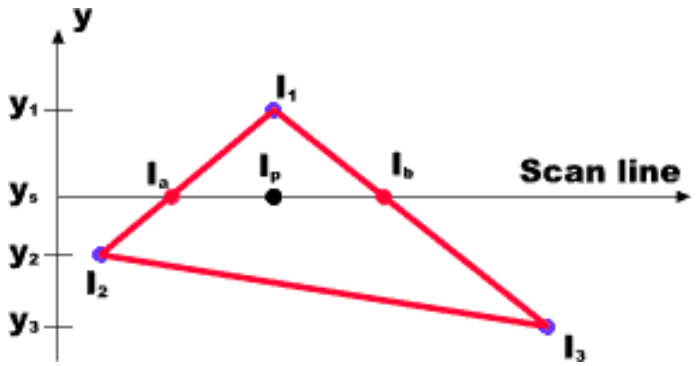


Benötigte Arithmetik:

Skalarprodukt, Multiplikation, Addition, Potenzierung

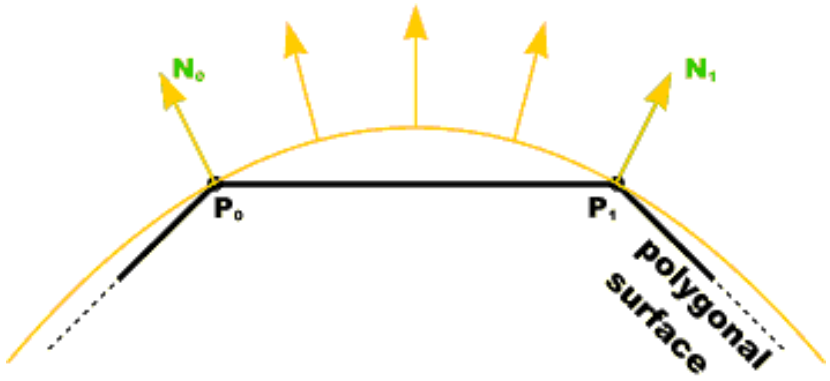
Schattierung

Gouraud



- Interpolation der Intensitäten
- Einfache Berechnung
- Hohe Geschwindigkeit

Phong



- Interpolation der Normalen
- Normalisierung pro Pixel(!)
- Evaluation des Beleuchtungs-modells pro Pixel(!)
- Hoher Rechenaufwand

GPU

Bisher: • Transformation/Schattierung auf CPU

• Übergabe der transformierten/schattierten Polygon an die Grafikhardware

Besser: Effiziente *Custom-Chips* für Transformation und Schattierung

Graphics **P**rocessing **U**nit

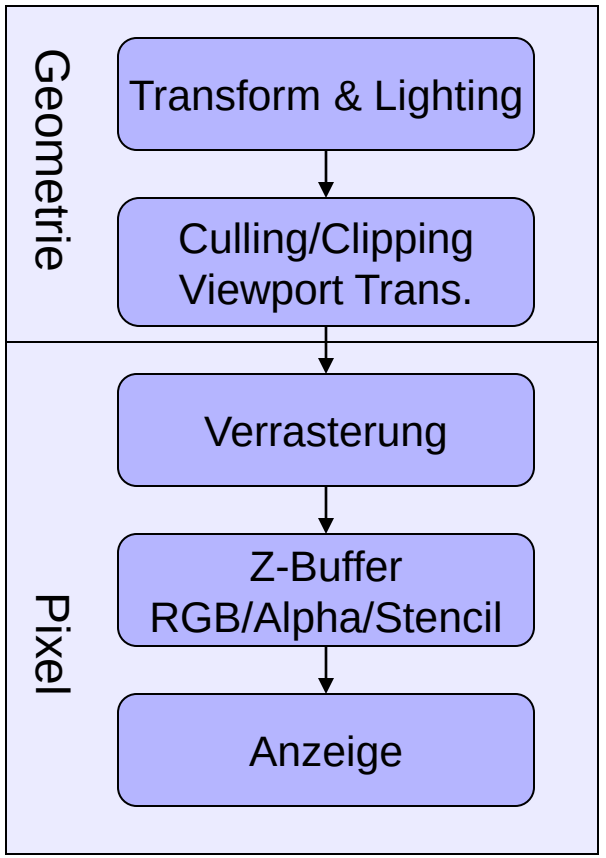
• Entlastung der CPU

• Höhere Effizienz als *general purpose* CPUs

GPU

Eingangsdaten:

- *Strom* aus Koordinaten / Normalen
- Lichtquellen
- Transformationen



⇨ Stream Processing

⇨ SIMD Architektur (*Unit*)

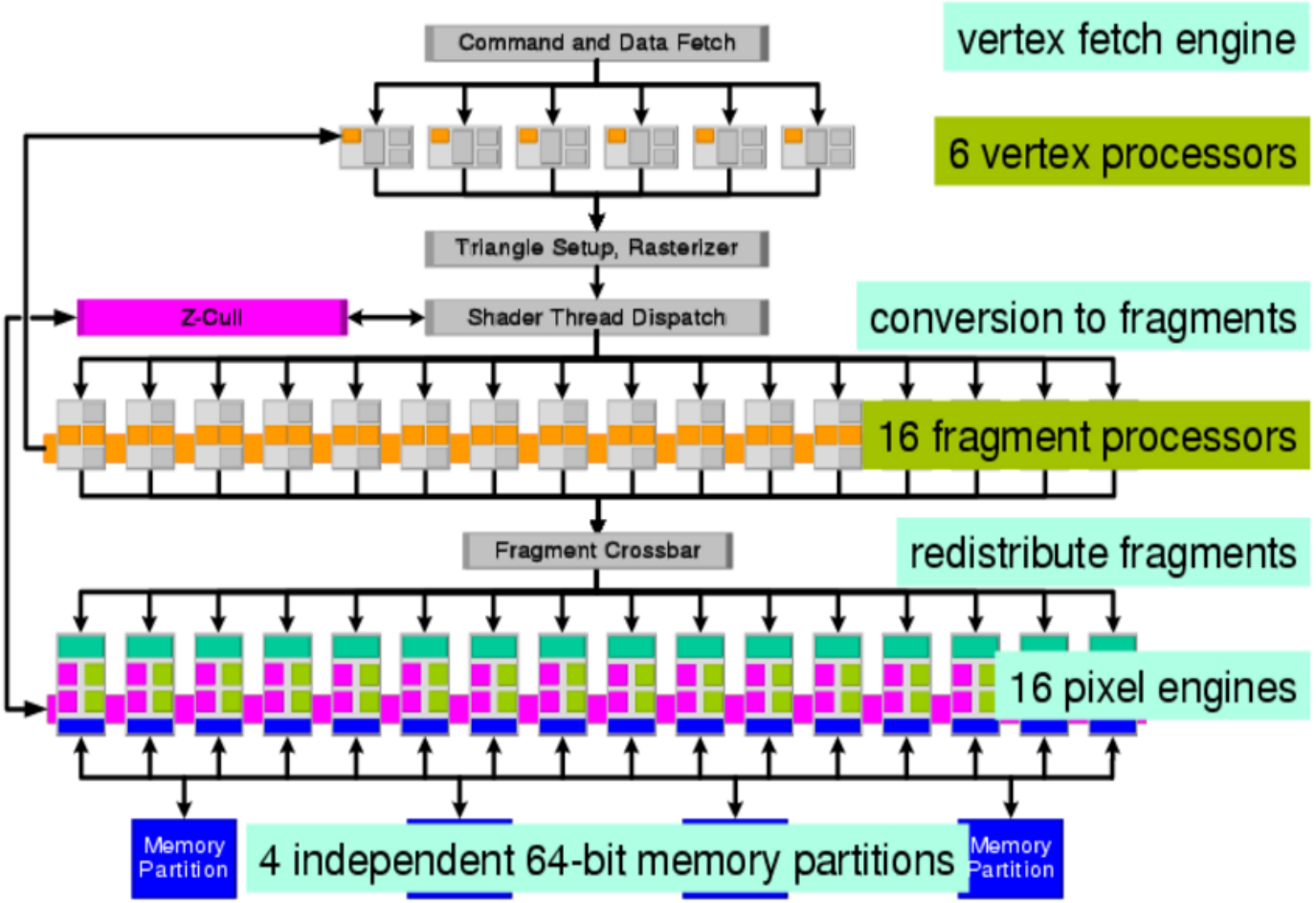
⇨ Mehrere *Units* parallel (MIMD)

⇨ Verstecken von Latenzzeiten durch *Pipelining*

NVIDIA GeForce 6800

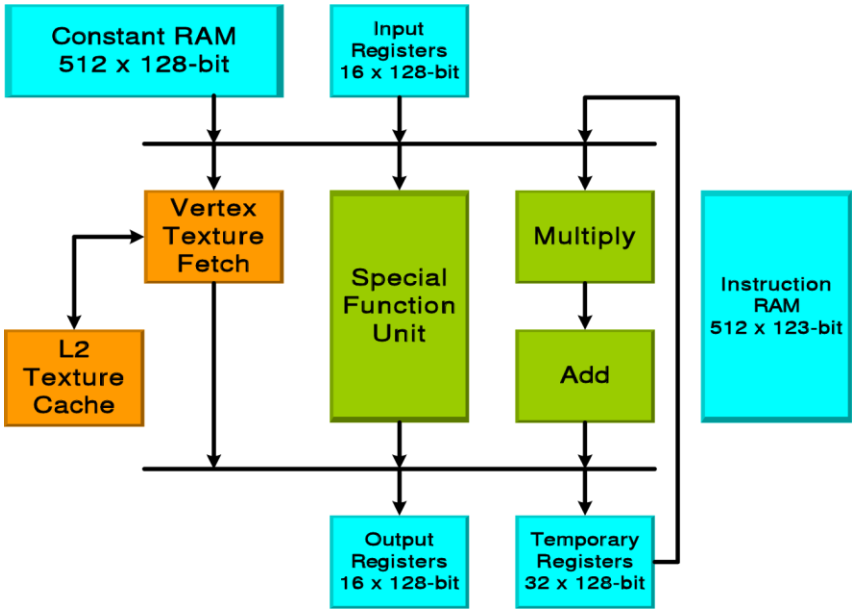
- 222 Mio. Transistoren (P4 Extreme Ed.: 188 Mio.)
- 400+ MHz GPU-Takt (üblich: ca. 425 MHz)
- 550 MHz RAM (256-Bit Bus, PC dual-channel: 128-Bit)
- ~ 120 GFLOP/s peak performance
- ~ 35GB/s memory bandwidth
- 6 Vertex Units
- 16 Fragment Units

GeForce 6800



GeForce 6800 Vertex Unit

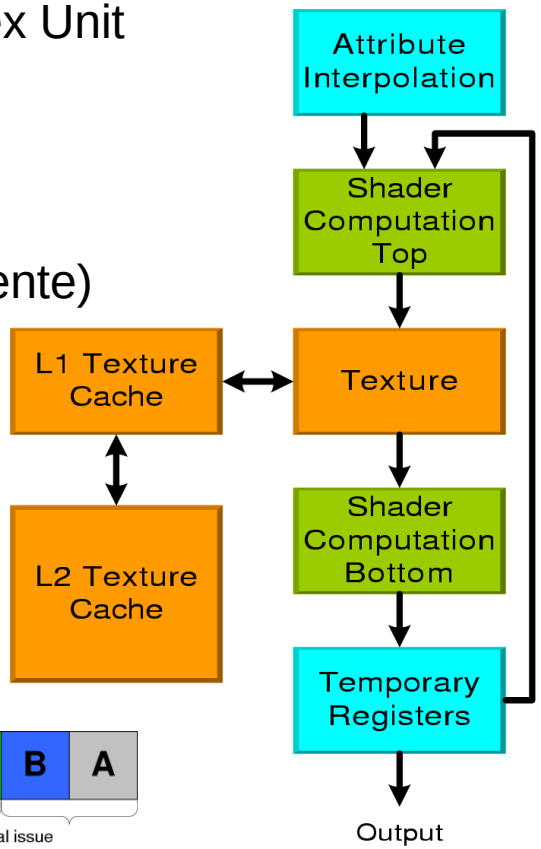
- 6 Vektor Fließkommaprozessoren (6 Vertex Shader Units)
  - VLIW Architektur
  - 512 x 128-Bit Context RAM
  - 32 x 128-Bit Register
  - 16 x 128-Bit In/Out Register
  - Platz für 512 Instruktionen
  - 32-Bit Floating-Point



- Bis 3 Threads pro Vertex Unit
  - Verstecken der Latenz

GeForce 6800 Fragment Unit

- Registersatz vergleichbar mit Vertex Unit
- VLIW / SIMD
- DSP-Befehle (z.B. MACs)
- 32-Bit Floating-Point (pro Komponente)
- Insgesamt 16 Units
- max. 6 Instruktionen parallel



Shader Unit 1
4 FP Ops / pixel
Dual/Co-Issue
Texture Address Calc
Free fp16 normalize
+ mini ALU

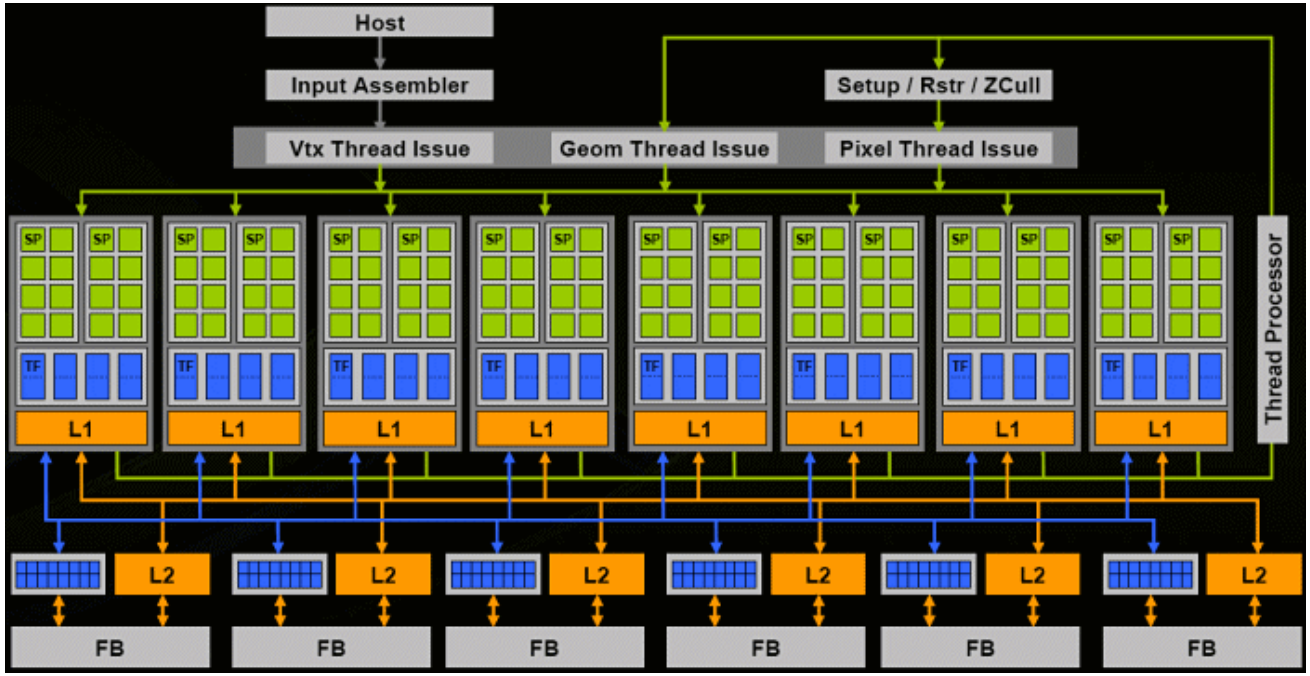
Texture Filter
Bi / Tri / Aniso
1 texture @ full speed
4-tap filter @ full speed
16:1 Aniso w/ Trilinear (128-tap)
FP16 Texture Filtering

Shader Unit 2
4 FP Ops / pixel
Dual/Co-Issue
+ mini ALU

NVIDIA GeForce 8800 (G80)

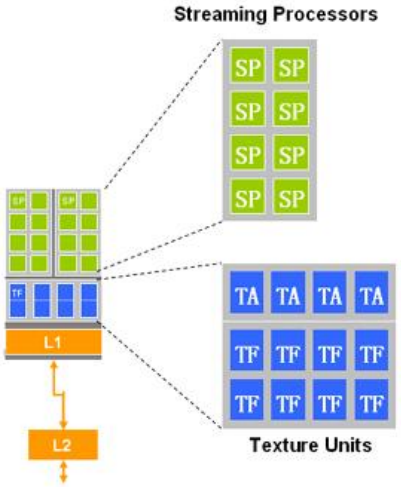
- 680 Mio. Transistoren (P4 Extreme Ed.: 188 Mio.)
- 1.35 GHz GPU-Takt
- 575 MHz RAM-Takt (384-Bit Bus, PC dual-channel: 128-Bit)
- ~ 520 GFLOP/s peak performance
- ~ 86 GB/s memory bandwidth
- 8 Gruppen von je 16 Streamprozessoren

GeForce 8800 (G80)



- Unifiziertes Design (Units können für versch. Shadertypen eingesetzt werden)
- Kürzere Pipelines
- Mehr Threads

GeForce 8800 (G80)



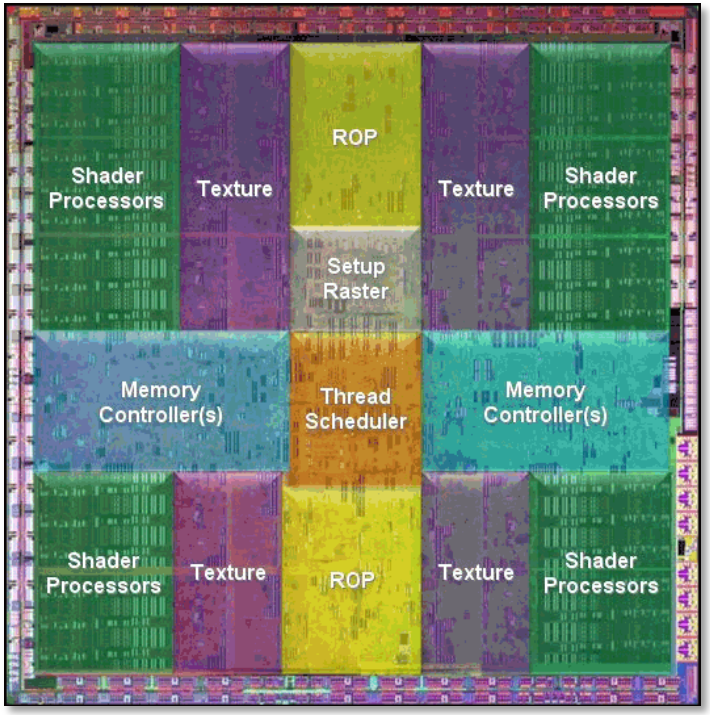
- **SP = Streaming Processors**
- **TF = Texture Filtering Unit**
- **TA = Texture Address Unit**
- **L1/L2 = Caches**

- Skalare Prozessoren
- Dual-issue MAD/MUL
- Gruppierung zu Einheiten
- Verteilung auf Workload

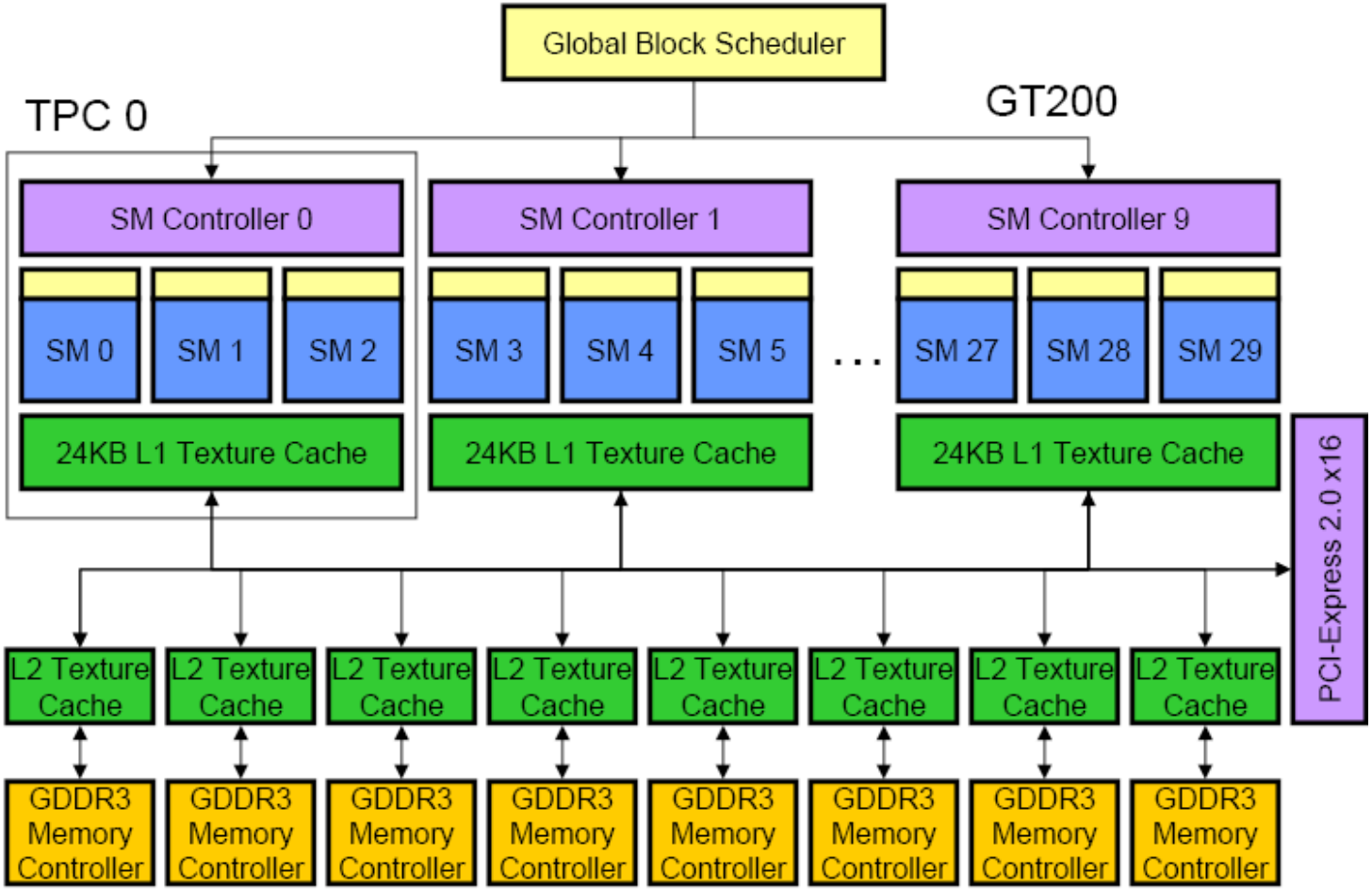
- Umstieg auf skalare Prozessoren bedingt durch Auswertung von Shaderprogrammen
- Bessere Auslastung der Shader-Einheiten
- 2-fache Geschwindkeitssteigerung (lt. nVidia)

NVIDIA GT200

- 1.4 Mrd. Transistoren
- 1.4 GHz GPU-Takt
- 1.2 MHz RAM-Takt (512-Bit Bus)
- ~ 1 TFLOP/s peak performance
- ~ 160 GB/s memory bandwidth
- 10 Gruppen von je 24 Streamprozessoren



GT200


Global Block Scheduler
TPC 0
GT200
SM Controller 0
SM Controller 1
SM Controller 9
SM 0
SM 1
SM 2
SM 3
SM 4
SM 5
SM 27
SM 28
SM 29
24KB L1 Texture Cache
24KB L1 Texture Cache
24KB L1 Texture Cache
PCI-Express 2.0 x16
L2 Texture Cache
L2 Texture Cache
L2 Texture Cache
L2 Texture Cache
L2 Texture Cache
L2 Texture Cache
L2 Texture Cache
L2 Texture Cache
GDDR3 Memory Controller
GDDR3 Memory Controller
GDDR3 Memory Controller
GDDR3 Memory Controller
GDDR3 Memory Controller
GDDR3 Memory Controller
GDDR3 Memory Controller
GDDR3 Memory Controller

Programmiermodell

Stream Processing

Stream: (geordneter) Datenstrom von einfachen oder auch komplexen Datentypen

Kernel: Programm, das Berechnungen auf einem Stream ausführt

Eigenschaften:

- Keine Datenabhängigkeiten zwischen den Stream-Elementen
- Eingabe ist *read-only*
- Ausgabe ist *write-only*
- Eingeschränkter Kontrollfluss

Stream Processing

Konsequenzen:

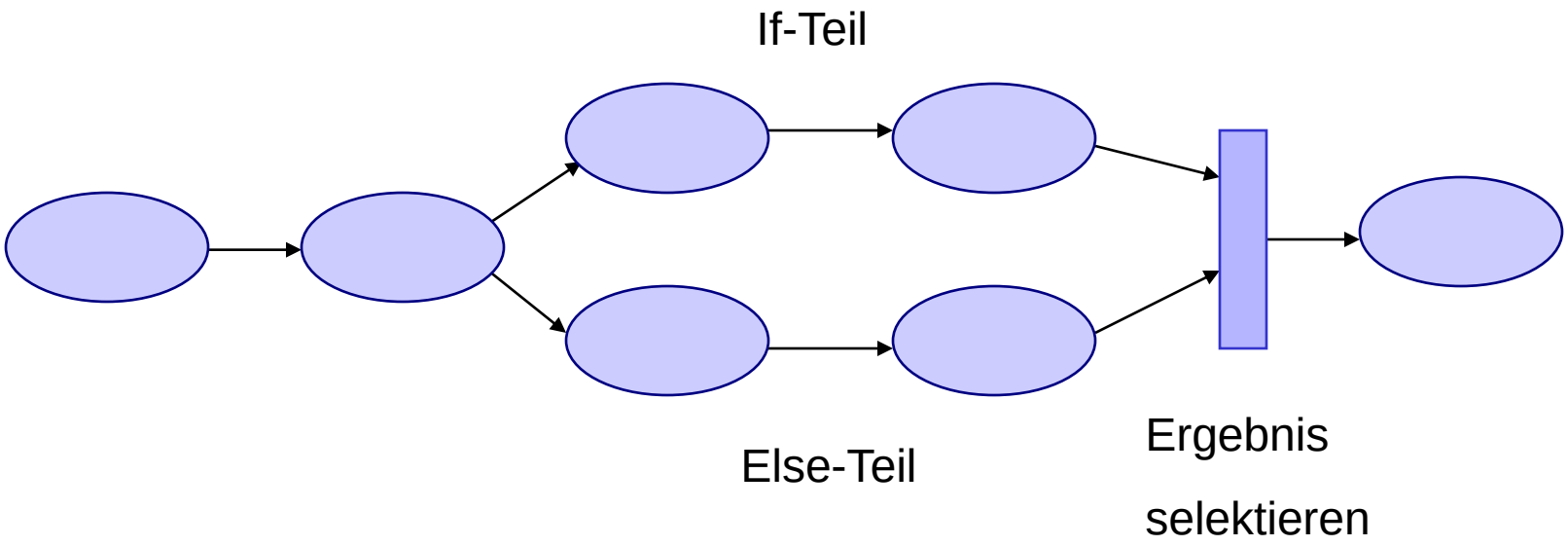
- Hohe Datenparallelität (keine Abhängigkeiten im Stream)
- Tiefe Pipelines (Verketten von Kernels)
- Hohe Lokalität (Register / Caching)
- **Eingeschränkte Programmierung**

Beispiel: Verzweigung (if-then-else Anweisungen)

- grundsätzlich vermeiden
- in einigen Architekturen nicht ohne weiteres möglich

Stream Processing

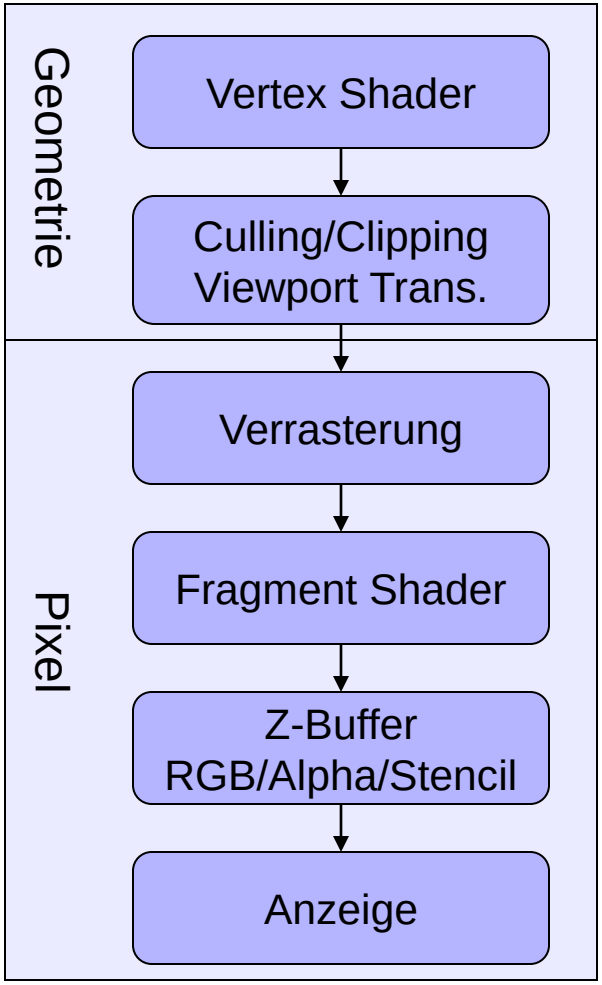
Möglicher Workaround: if **und** else Teil ausführen



- Moderne GPUs unterstützen *dynamic branching*

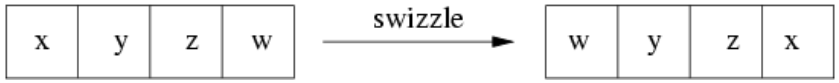
Programmierbare Shader

- Erhöhung der Flexibilität
- Verfeinerung der Schattierung
  - Komplexe Beleuchtungsmodelle (z.B. Anisotropie)
  - BRDFs
- Verbesserung der Texturierung
  - Texturverarbeitung / Bumpmapping / prozedurale Texturen
- Effekte/Animationen in Echtzeit



Shaderprogrammierung

- Vektorarithmetik (4-Komponenten Vektor, vgl. Intel SSE)
- Horizontale und Vertikale Arithmetik
  - Skalarprodukt
  - Vektoraddition
  - Matrix/Vektor Multiplikation
  - ...
- Swizzling:



- Writemask

GL Shading Language (GLSL)

- Hochsprache für Shader-Programmierung
- Syntax sehr C-ähnlich
- Vertex- und Fragmentprogramme
- Source wird zur Laufzeit compiliert und gelinked
- Parameterübergabe vom Vertex ins Fragmentprogramm möglich

GLSL Beispiel (Fragmentshader)

- Einfaches Beispiel für Volumenvisualisierung (Farb-LUT):

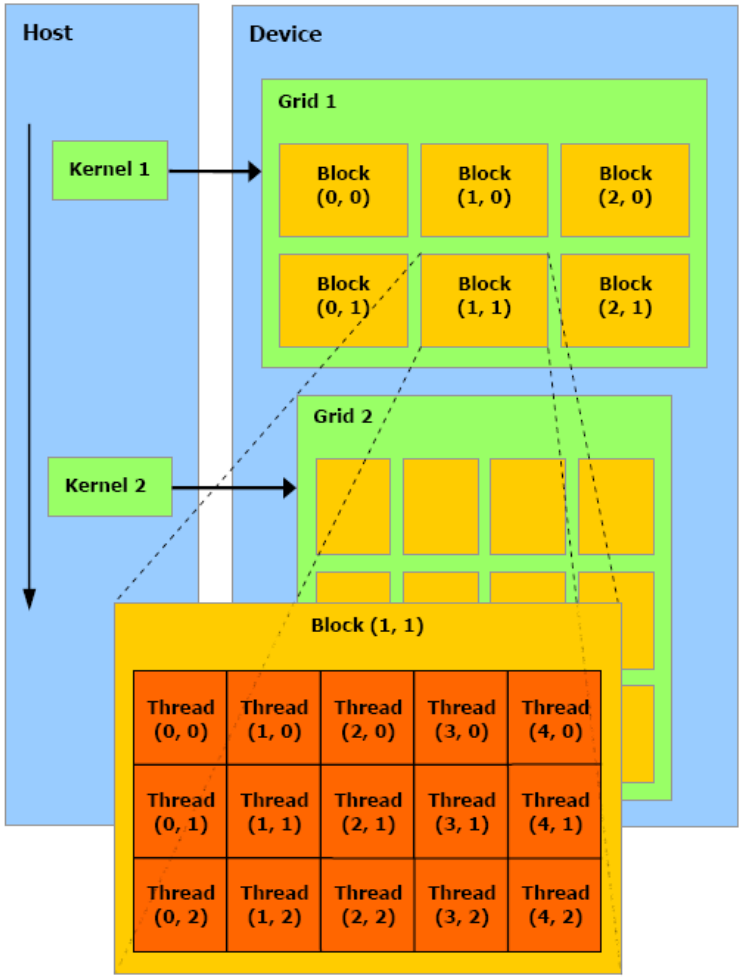
```
uniform sampler3D VolumeTexture;
uniform sampler2D TransferTable;

void main() {
  float voxel;
  vec2 transferindex;
  vec4 ambient;
  voxel=texture3D(VolumeTexture,gl_TexCoord[0].xyz).r;
  transferindex=vec2(voxel,0.0);
  ambient=texture2D(TransferTable,transferindex);
  gl_FragColor=ambient;
}
```

Compute Unified Device Architecture

- Eingeführt mit GeForce 8xxx (G80)
- Framework für allg. Stream-Programming
- Arbeitet mit hohen Zahlen an Threads
- Syntax angelehnt an C

CUDA

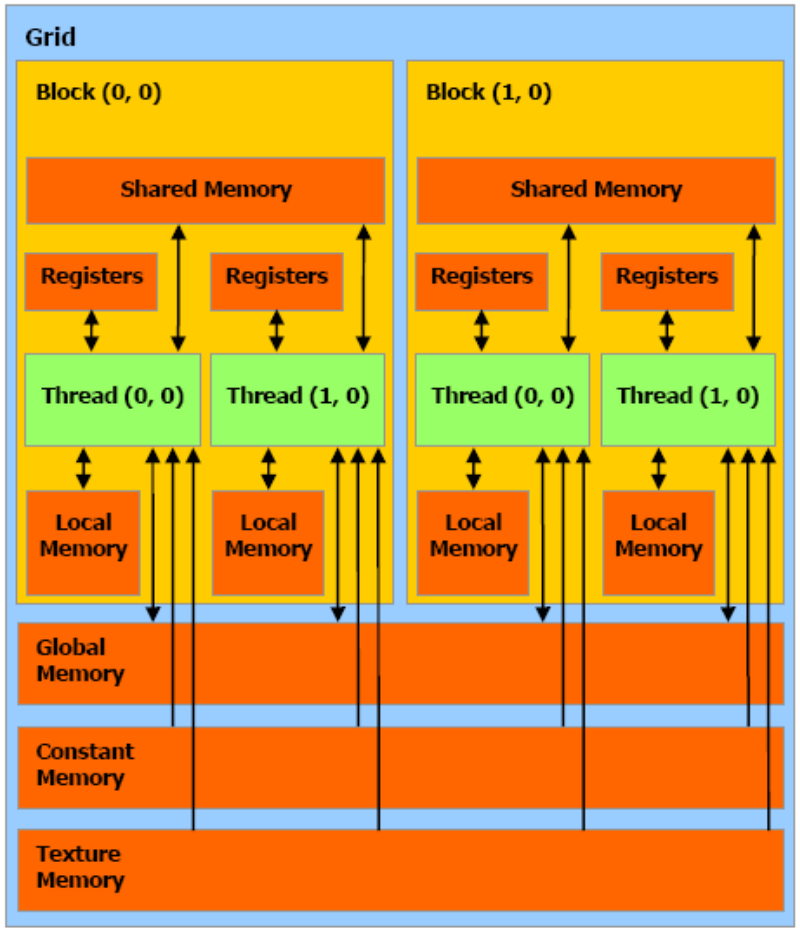
- Programme: *Kernels*
- Zusammenfassung in Thread-Blocks
- Threads in einem Block teilen sich Daten
- Kommunikation/Synchronisation nur innerhalb desselben Blockes
- Blöcke werden in *Grids* zusammengefasst



CUDA

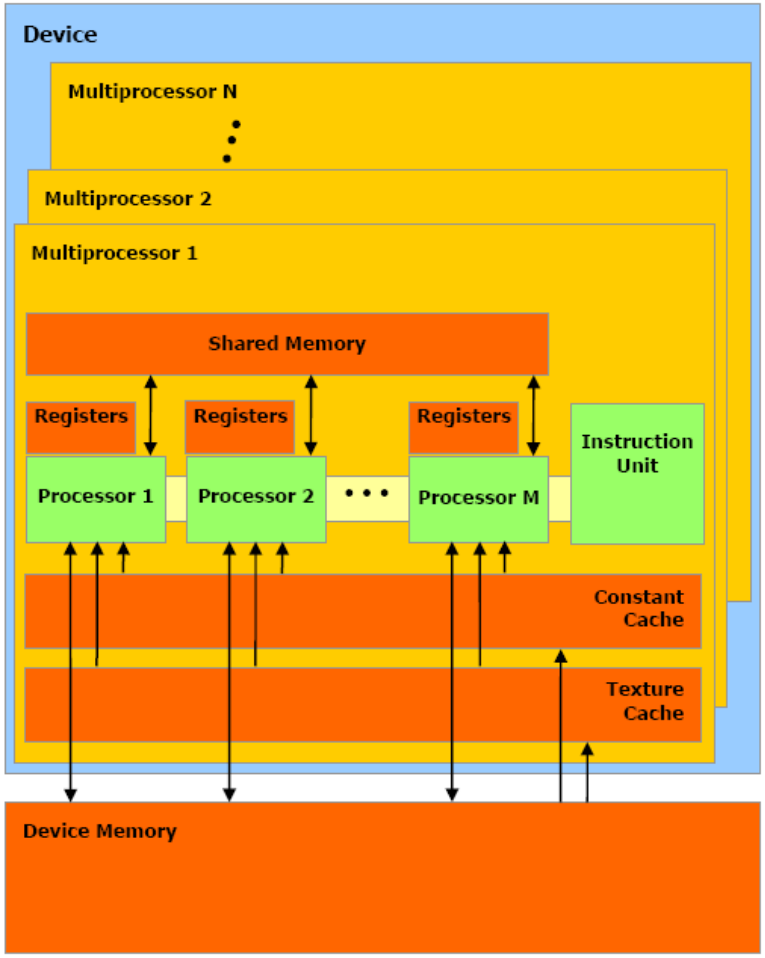
Scheduler führt Threads in Blöcken (warps) aus (32 Threads)

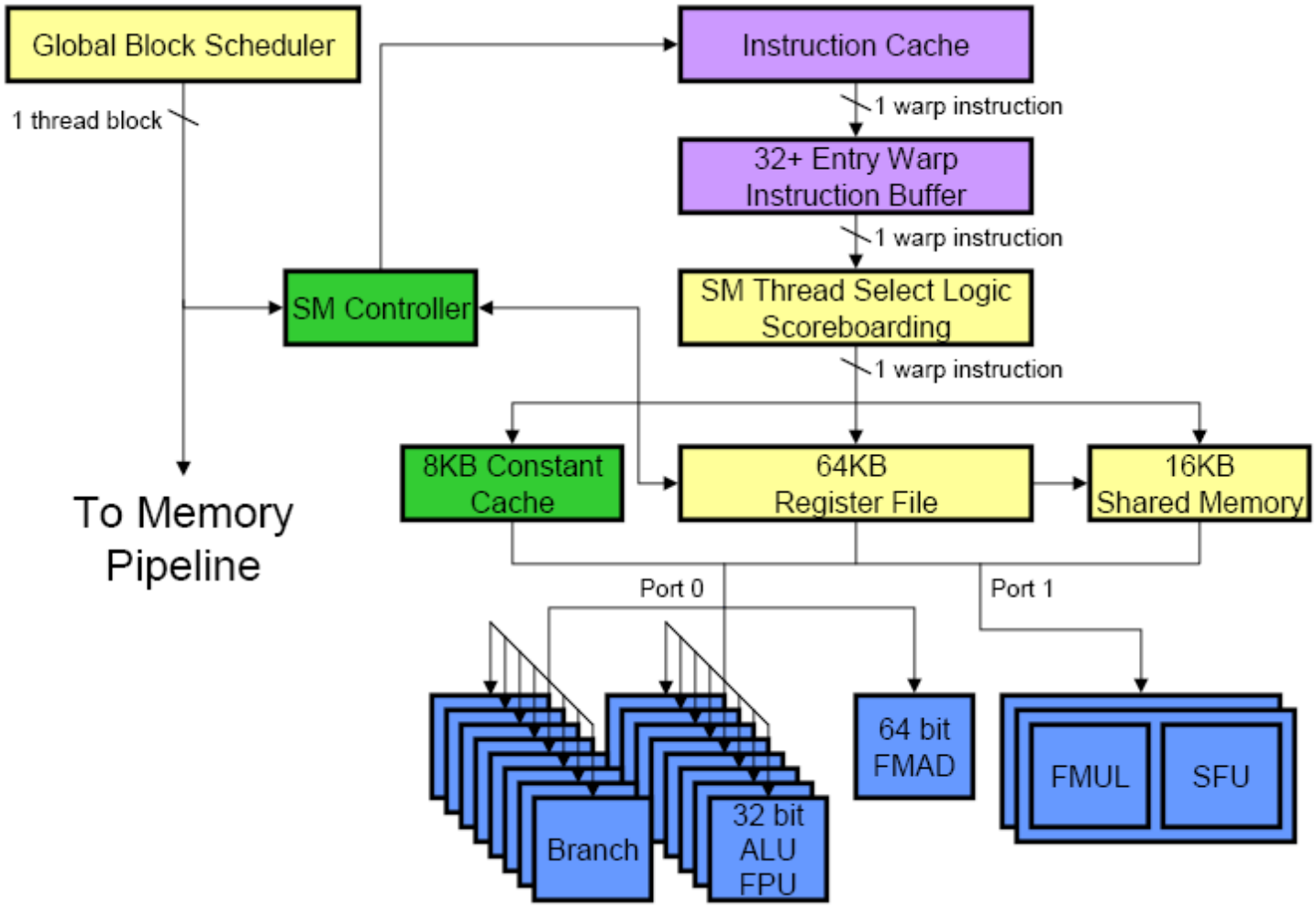
- Dynamic register allocation
- Reihenfolge der *warp* Ausführung nicht definiert
- Synchronisation innerhalb eines Blocks programmatisch möglich
- 8 Prozessoren = 1 Multiprozessor
- *Local Memory* ist Teil des Framebuffers
- *Shared Local* Memory ist *on-chip*



CUDA

- Zusammenfassen der Prozessoren zu *Multiprozessoren*
- Gemeinsame Caches für jeweils einen Multiprozessor
- Alle Einzelprozessoren eines MP führen denselben Befehl aus (SIMT)
- Ähnlichkeit zu SIMD




Global Block Scheduler
1 thread block
To Memory Pipeline
Instruction Cache
1 warp instruction
32+ Entry Warp Instruction Buffer
1 warp instruction
SM Controller
SM Thread Select Logic Scoreboarding
1 warp instruction
8KB Constant Cache
64KB Register File
16KB Shared Memory
Port 0
Port 1
Branch
32 bit ALU FPU
64 bit FMAD
FMUL
SFU

CUDA

Performance (GeForce 8800GTX):

- Multiply bzw. Multiply-Add: 4 Takte pro warp
- RCP/RCPSQRT/Log: 16 Takte pro warp
- FP Division: 20-36 Takte pro warp
- Bedingt *dual-issue* fähig (z.B. MAD+MUL simultan)

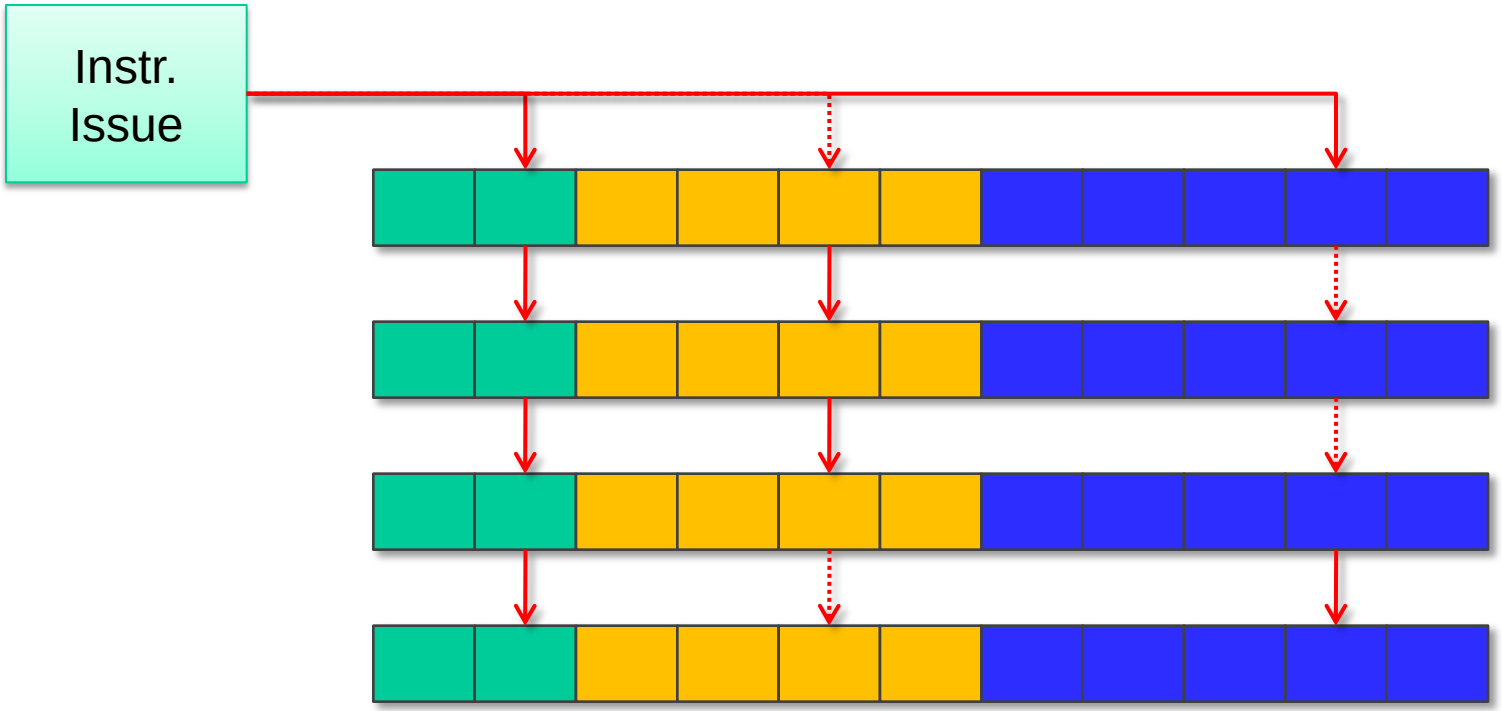
Peak Performance: 3*(32/4)*30*1.4 = **1,008 TFLOP/s**

BLAS3/SP Operationen (Matrix/Matrix Multiplikation):

- GF8800: 100+ GFLOP/s
- Cell BE: 175 GFLOP/s
- GT200: 370 GFLOP/s

CUDA

- Verzweigungen senken Effizienz
- *Predicated Execution*



CUDA (Beispiel)

```
__global__ void cudaTest(float *array, int arraySize) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<arraySize) array[idx] = (array[idx] * array[idx])/4.0f;
}

int main(void)  {
  float *cpuArray,deviceArray;
  const int arraySize = 10;
  cpuArray = (float *)malloc(arraySize*sizeof(float));
  cudaMalloc((void **)&deviceArray,arraySize*sizeof(float));
  for (int i=0; i<arraySize; i++) cpuArray[i] = (float)i;
  cudaMemcpy(deviceArray,cpuArray,arraySize*sizeof(float),cudaMemcpyHostToDevice);

  int blockSize = 4;
  int numBlocks = arraySize/blockSize + (arraySize%blockSize == 0 ? 0:1);
  cudaTest <<< numBlocks,blockSize >>> (deviceArray,arraySize);
  cudaMemcpy(cpuArray,deviceArray,arraySize* sizeof(float), cudaMemcpyDeviceToHost);

  free(cpuArray);
  cudaFree(deviceArray);
}
```

Open Computing Language

- Standardisierte Stream-Programming Sprache
- Starke Ähnlichkeit zu CUDA
- Generiert Code für breitere Basis
  - CPU (Intel,IBM Cell)
  - DSP
  - GPU (ATI/NVIDIA/Intel)
- Unterstützt auch „native“ Kernel (z.B. CUDA)

Entwicklung

- Ausbau der Parallelität (mehr Units)
- Erhöhung der RAM-Geschwindigkeit
- Spezielle GPU Cluster für HPC
- Derzeitige Entwicklung der Grafikhardware (noch) schneller als Entwicklung der CPUs

Weitere Informationen

- Vorlesung „Mensch-Maschine Interaktion“

`http://www.opengl.org`

`http://developer.nvidia.com`

`http://www.ati.com`

`http://www.gpgpu.org`

`http://khronos.org/opencl`