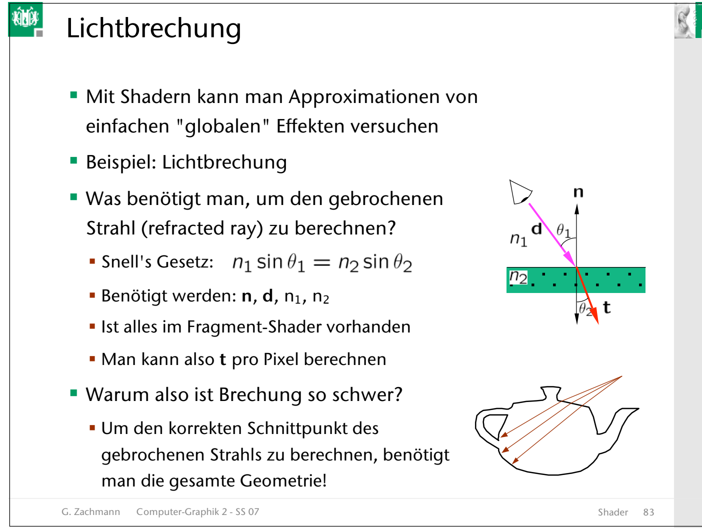
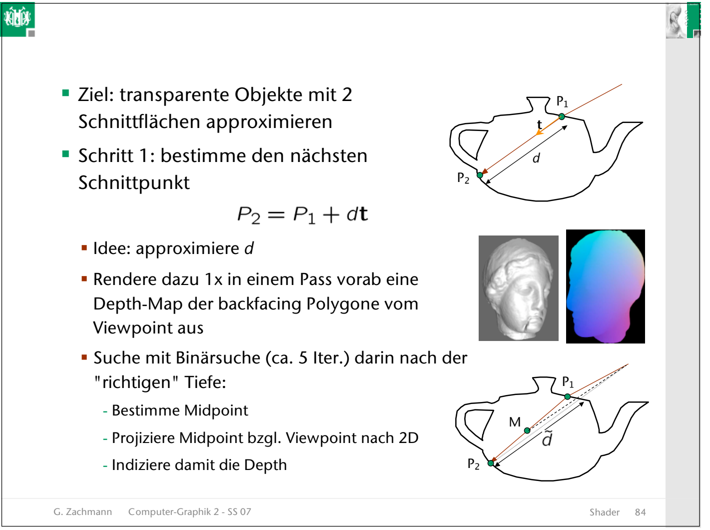
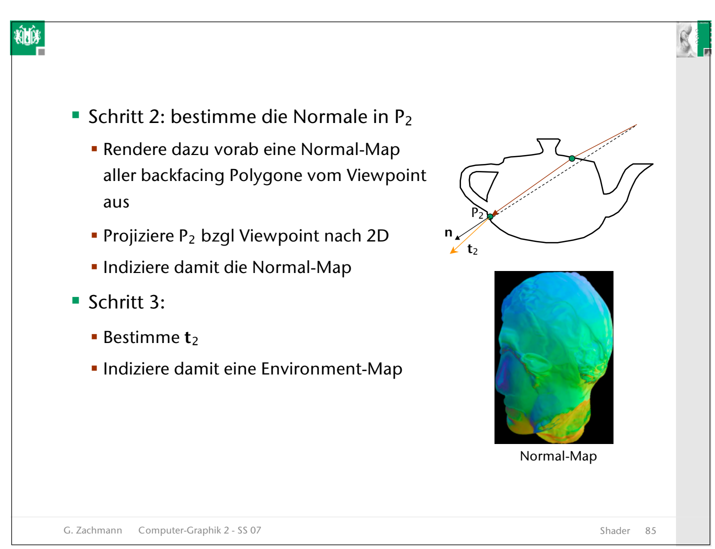
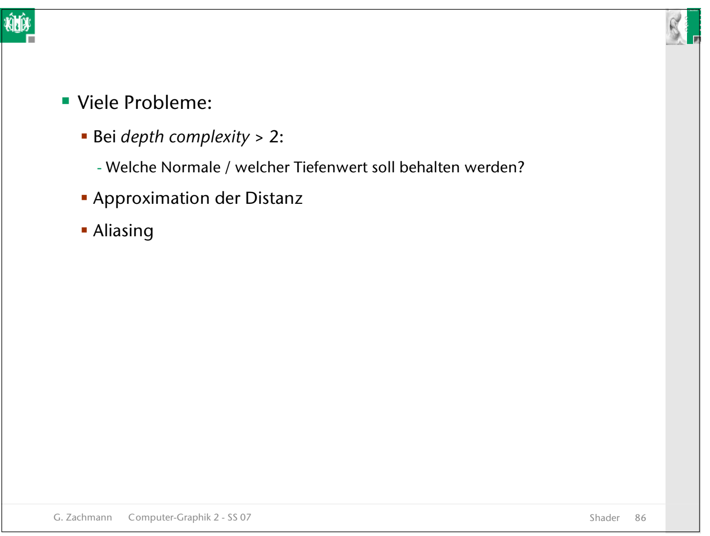
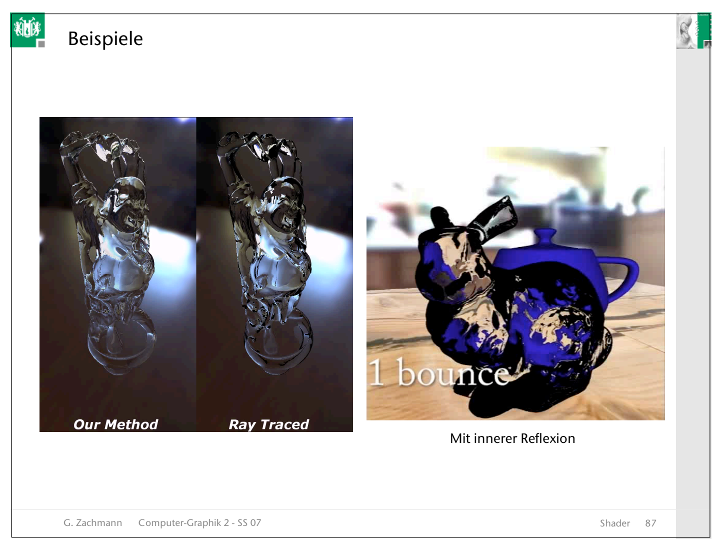
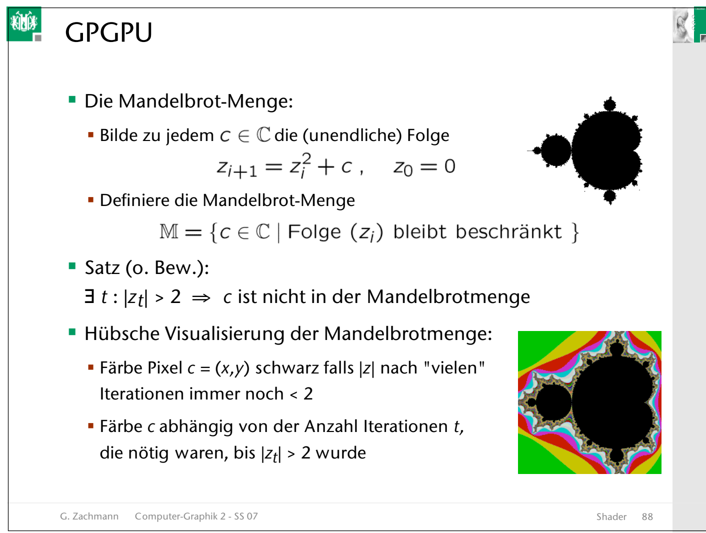

Lichtbrechung
Mit Shadern kann man Approximationen von einfachen "globalen" Effekten versuchen
Beispiel: Lichtbrechung
Was benötigt man, um den gebrochenen Strahl (refracted ray) zu berechnen?
Snell's Gesetz: $n_1 \sin\theta_1 = n_2 \sin\theta_2$
Benötigt werden: **n**, **d**, n_1, n_2
Ist alles im Fragment-Shader vorhanden
Man kann also **t** pro Pixel berechnen
Warum also ist Brechung so schwer?
Um den korrekten Schnittpunkt des gebrochenen Strahls zu berechnen, benötigt man die gesamte Geometrie!
n
d
n_1
θ_1
n_2
θ_2
t
G. Zachmann Computer-Graphik 2 - SS 07
Shader 83

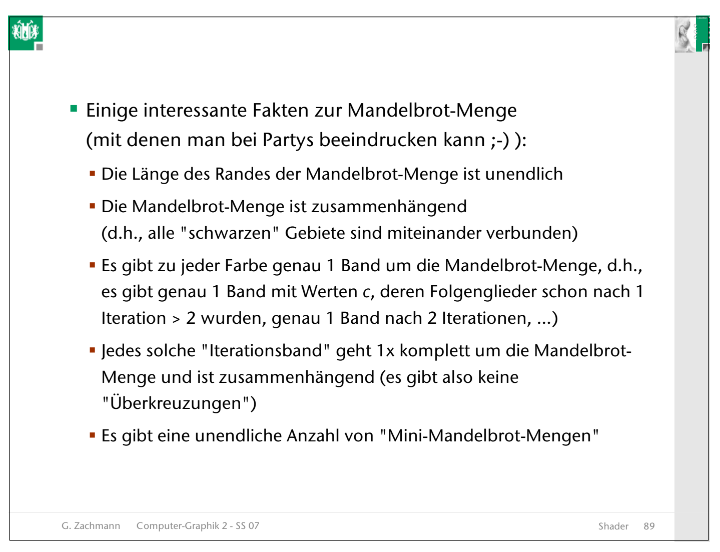

Ziel: transparente Objekte mit 2 Schnittflächen approximieren
Schritt 1: bestimme den nächsten Schnittpunkt
$P_2 = P_1 + d\mathbf{t}$
Idee: approximiere *d*
Rendere dazu 1x in einem Pass vorab eine Depth-Map der backfacing Polygone vom Viewpoint aus
Suche mit Binärsuche (ca. 5 Iter.) darin nach der "richtigen" Tiefe:
- Bestimme Midpoint
- Projiziere Midpoint bzgl. Viewpoint nach 2D
- Indiziere damit die Depth
P_1
t
d
P_2
M
$\tilde{d}$
G. Zachmann Computer-Graphik 2 - SS 07
Shader 84

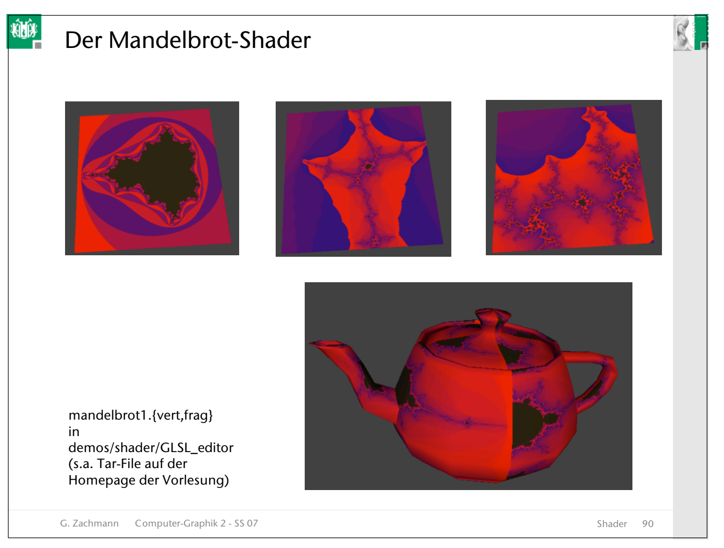

Schritt 2: bestimme die Normale in P_2
Rendere dazu vorab eine Normal-Map aller backfacing Polygone vom Viewpoint aus
Projiziere P_2 bzgl Viewpoint nach 2D
Indiziere damit die Normal-Map
Schritt 3:
Bestimme $\mathbf{t}_2$
Indiziere damit eine Environment-Map
P_2
n
$\mathbf{t}_2$
Normal-Map
G. Zachmann Computer-Graphik 2 - SS 07
Shader 85

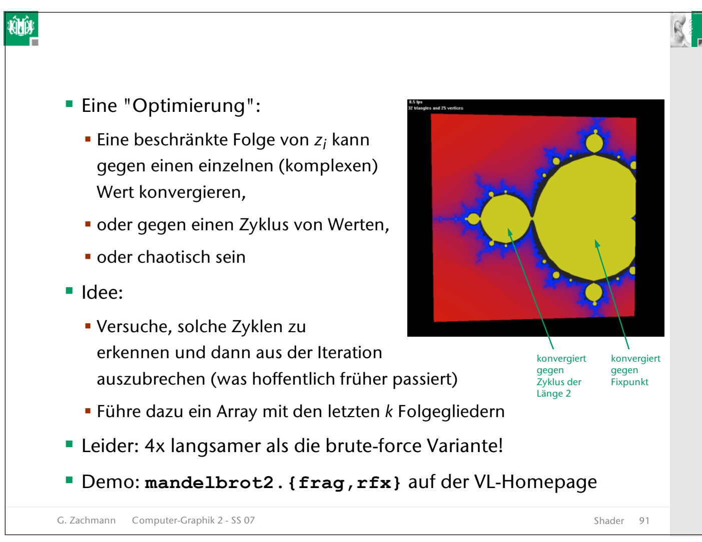

Viele Probleme:
Bei *depth complexity* > 2:
- Welche Normale / welcher Tiefenwert soll behalten werden?
Approximation der Distanz
Aliasing
G. Zachmann Computer-Graphik 2 - SS 07
Shader 86

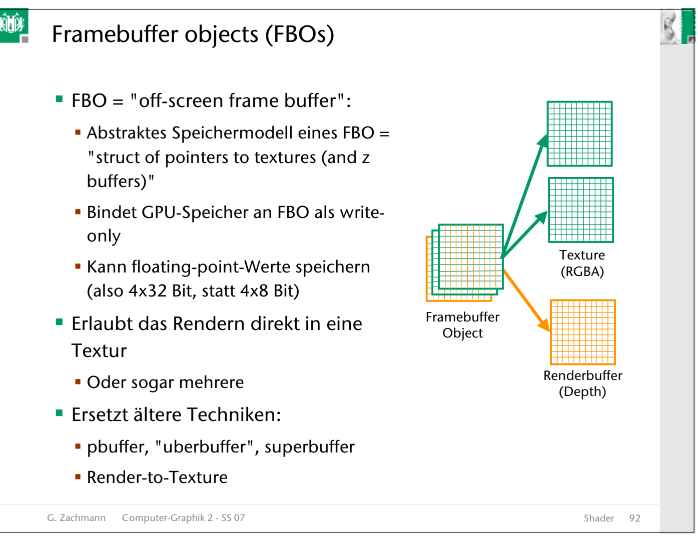

Beispiele
Our Method
Ray Traced
1 bounce
Mit innerer Reflexion
G. Zachmann Computer-Graphik 2 - SS 07
Shader 87

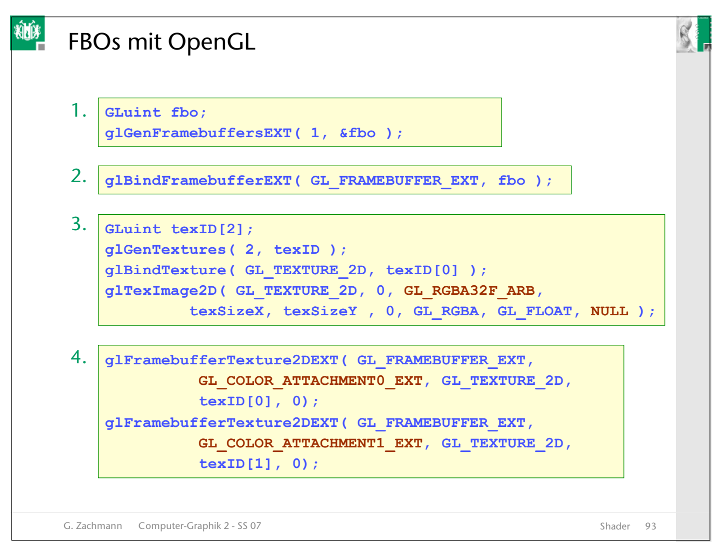

GPGPU
Die Mandelbrot-Menge:
Bilde zu jedem $c \in \mathbb{C}$ die (unendliche) Folge
$z_{i+1} = z_i^2 + c\,, \quad z_0 = 0$
Definiere die Mandelbrot-Menge
$\mathbb{M} = \{ c \in \mathbb{C} \mid \text{Folge } (z_i) \text{ bleibt beschränkt } \}$
Satz (o. Bew.):
$\exists\, t : |z_t| > 2 \Rightarrow c$ ist nicht in der Mandelbrotmenge
Hübsche Visualisierung der Mandelbrotmenge:
Färbe Pixel $c = (x,y)$ schwarz falls $|z|$ nach "vielen" Iterationen immer noch < 2
Färbe c abhängig von der Anzahl Iterationen t, die nötig waren, bis $|z_t| > 2$ wurde
G. Zachmann Computer-Graphik 2 - SS 07
Shader 88


Einige interessante Fakten zur Mandelbrot-Menge (mit denen man bei Partys beeindrucken kann ;-)):
Die Länge des Randes der Mandelbrot-Menge ist unendlich
Die Mandelbrot-Menge ist zusammenhängend (d.h., alle "schwarzen" Gebiete sind miteinander verbunden)
Es gibt zu jeder Farbe genau 1 Band um die Mandelbrot-Menge, d.h., es gibt genau 1 Band mit Werten c, deren Folgenglieder schon nach 1 Iteration > 2 wurden, genau 1 Band nach 2 Iterationen, ...)
Jedes solche "Iterationsband" geht 1x komplett um die Mandelbrot-Menge und ist zusammenhängend (es gibt also keine "Überkreuzungen")
Es gibt eine unendliche Anzahl von "Mini-Mandelbrot-Mengen"
G. Zachmann Computer-Graphik 2 - SS 07
Shader 89


Der Mandelbrot-Shader
mandelbrot1.{vert,frag}
in
demos/shader/GLSL_editor
(s.a. Tar-File auf der
Homepage der Vorlesung)
G. Zachmann Computer-Graphik 2 - SS 07
Shader 90



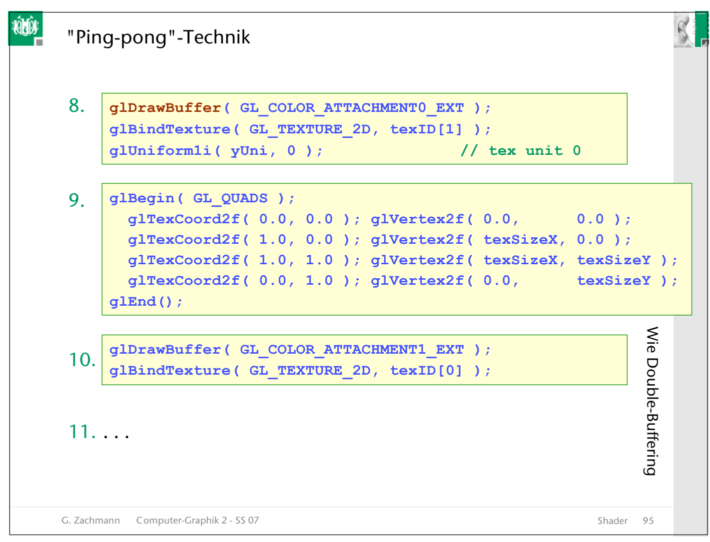


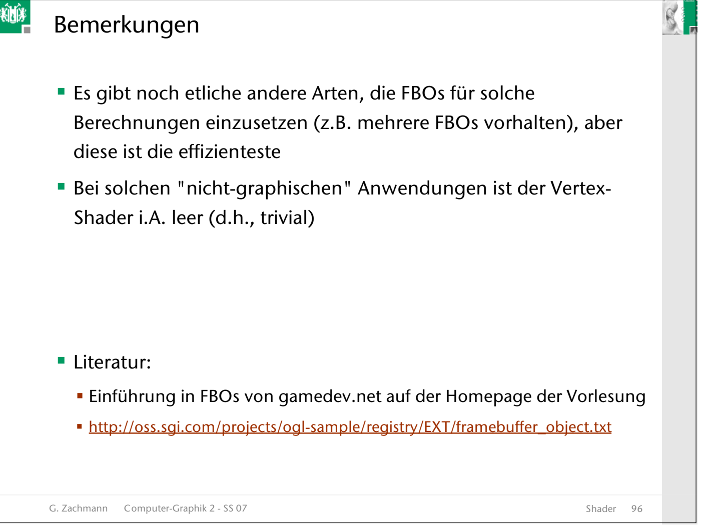


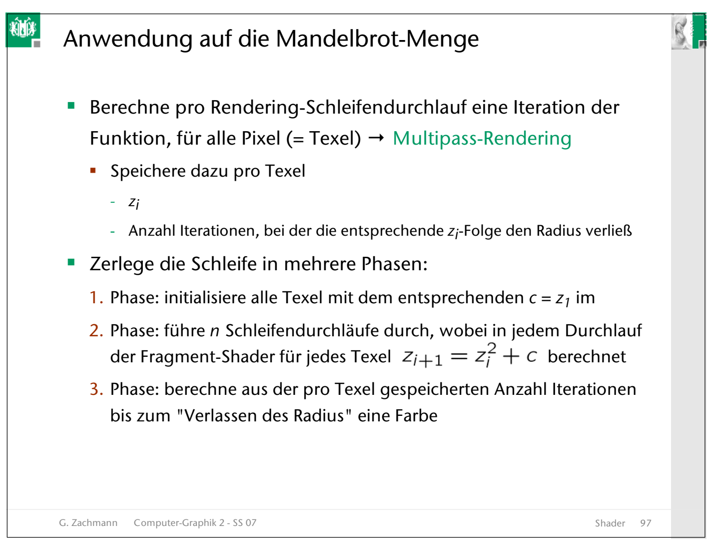
5.
```
GLenum status = glCheckFramebufferStatusEXT(
                                    GL_FRAMEBUFFER_EXT );
if ( status != GL_FRAMEBUFFER_COMPLETE_EXT ) ...
```

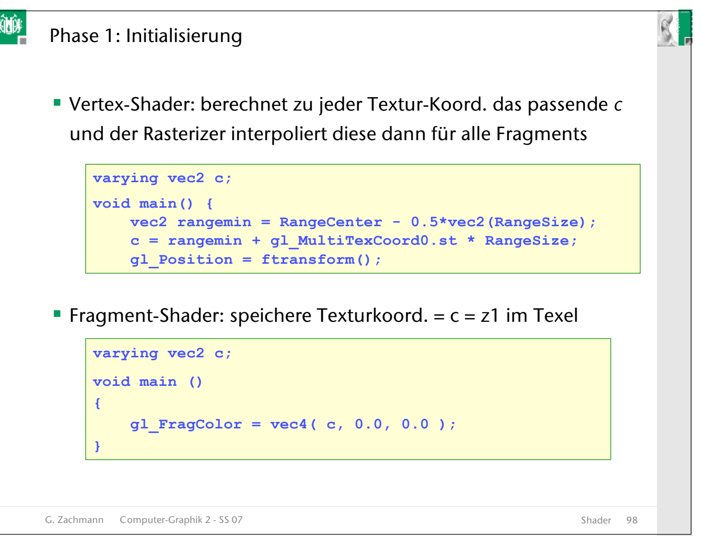
6.
```
glViewport( 0, 0, texSizeX, texSizeY );
glMatrixMode( GL_PROJECTION );   glLoadIdentity();
gluOrtho2D( 0.0, texSizeX, 0.0, texSizeY );
glMatrixMode( GL_MODELVIEW );    glLoadIdentity();
```

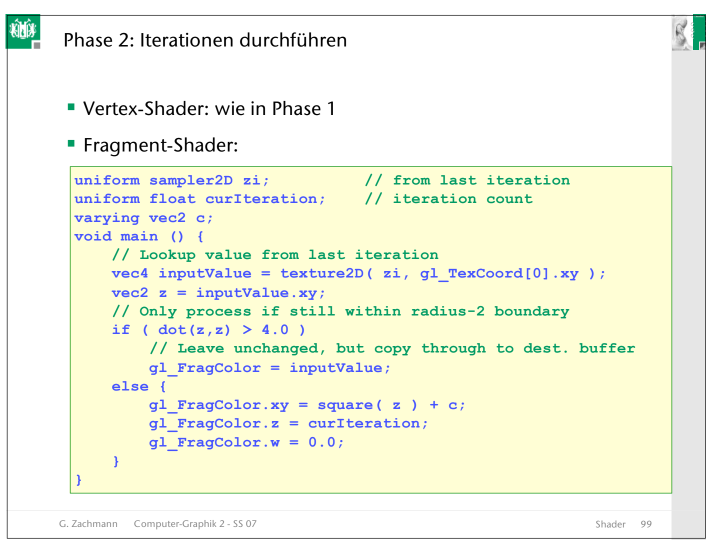
7.
```
sh_prog_id = setShaders( "m.vert", "m.frag" );
yUni = glGetUniformLocation( sh_prog_id, "textureY" );
```


"Ping-pong"-Technik
8.
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
glBindTexture(GL_TEXTURE_2D, texID[1]);
glUniform1i(yUni, 0); // tex unit 0
9.
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0); glVertex2f(0.0, 0.0);
glTexCoord2f(1.0, 0.0); glVertex2f(texSizeX, 0.0);
glTexCoord2f(1.0, 1.0); glVertex2f(texSizeX, texSizeY);
glTexCoord2f(0.0, 1.0); glVertex2f(0.0, texSizeY);
glEnd();
10.
glDrawBuffer(GL_COLOR_ATTACHMENT1_EXT);
glBindTexture(GL_TEXTURE_2D, texID[0]);
11. . . .
Wie Double-Buffering
G. Zachmann Computer-Graphik 2 - SS 07
Shader 95


Bemerkungen
Es gibt noch etliche andere Arten, die FBOs für solche Berechnungen einzusetzen (z.B. mehrere FBOs vorhalten), aber diese ist die effizienteste
Bei solchen "nicht-graphischen" Anwendungen ist der Vertex-Shader i.A. leer (d.h., trivial)
Literatur:
Einführung in FBOs von gamedev.net auf der Homepage der Vorlesung
http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt
G. Zachmann Computer-Graphik 2 - SS 07
Shader 96


Anwendung auf die Mandelbrot-Menge
Berechne pro Rendering-Schleifendurchlauf eine Iteration der Funktion, für alle Pixel (= Texel) → Multipass-Rendering
Speichere dazu pro Texel
zi
Anzahl Iterationen, bei der die entsprechende zi-Folge den Radius verließ
Zerlege die Schleife in mehrere Phasen:
1. Phase: initialisiere alle Texel mit dem entsprechenden c = z1 im
2. Phase: führe n Schleifendurchläufe durch, wobei in jedem Durchlauf der Fragment-Shader für jedes Texel zi+1 = zi2 + c berechnet
3. Phase: berechne aus der pro Texel gespeicherten Anzahl Iterationen bis zum "Verlassen des Radius" eine Farbe
G. Zachmann Computer-Graphik 2 - SS 07
Shader 97


Phase 1: Initialisierung
Vertex-Shader: berechnet zu jeder Textur-Koord. das passende c und der Rasterizer interpoliert diese dann für alle Fragments
varying vec2 c;
void main() {
vec2 rangemin = RangeCenter - 0.5*vec2(RangeSize);
c = rangemin + gl_MultiTexCoord0.st * RangeSize;
gl_Position = ftransform();
Fragment-Shader: speichere Texturkoord. = c = z1 im Texel
varying vec2 c;
void main ()
{
gl_FragColor = vec4(c, 0.0, 0.0);
}
G. Zachmann Computer-Graphik 2 - SS 07
Shader 98



Phase 2: Iterationen durchführen

- Vertex-Shader: wie in Phase 1
- Fragment-Shader:

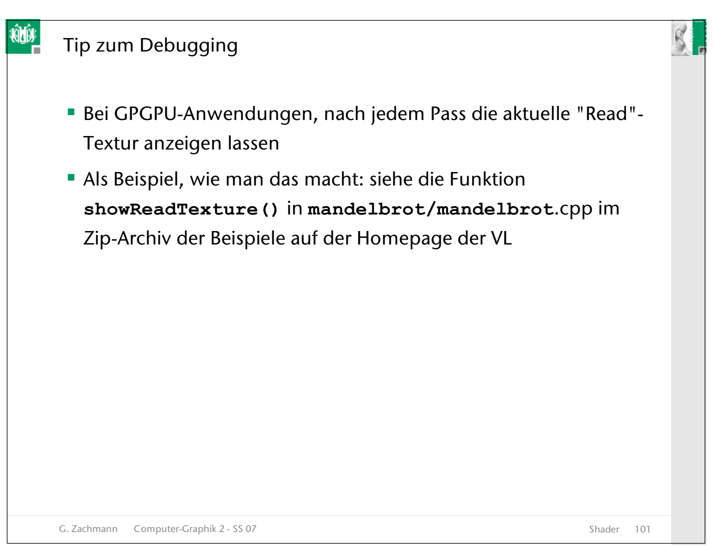
```
uniform sampler2D zi;          // from last iteration
uniform float curIteration;    // iteration count
varying vec2 c;
void main () {
    // Lookup value from last iteration
    vec4 inputValue = texture2D( zi, gl_TexCoord[0].xy );
    vec2 z = inputValue.xy;
    // Only process if still within radius-2 boundary
    if ( dot(z,z) > 4.0 )
        // Leave unchanged, but copy through to dest. buffer
        gl_FragColor = inputValue;
    else {
        gl_FragColor.xy = square( z ) + c;
        gl_FragColor.z = curIteration;
        gl_FragColor.w = 0.0;
    }
}
```


Phase 3: Iterationen in Farben umrechnen

4. Shader erstellt ein hübsches Falschfarbenbild:

```
uniform sampler2D input;        // from last iteration
uniform float maxIterations;
void main ()
{
    // Lookup value from last iteration
    vec4 inputValue = texture2D(input, gl_TexCoord[0].xy);
    vec2 z = inputValue.xy;
    float n = inputValue.z;
    if ( n < maxIterations )
        // compute gl_FragColor just as before
        // using inputValue.z / maxIterations
    else
        gl_FragColor = insideColor;
}
```

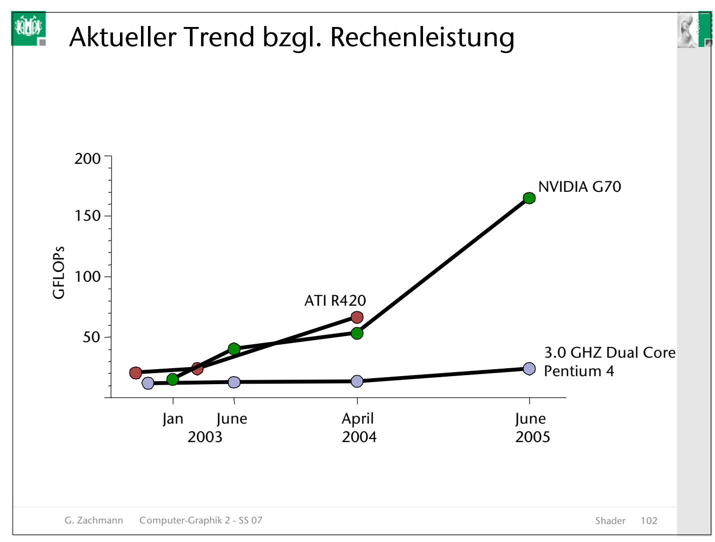
- Demo-Code: siehe Homepage der Vorlesung!

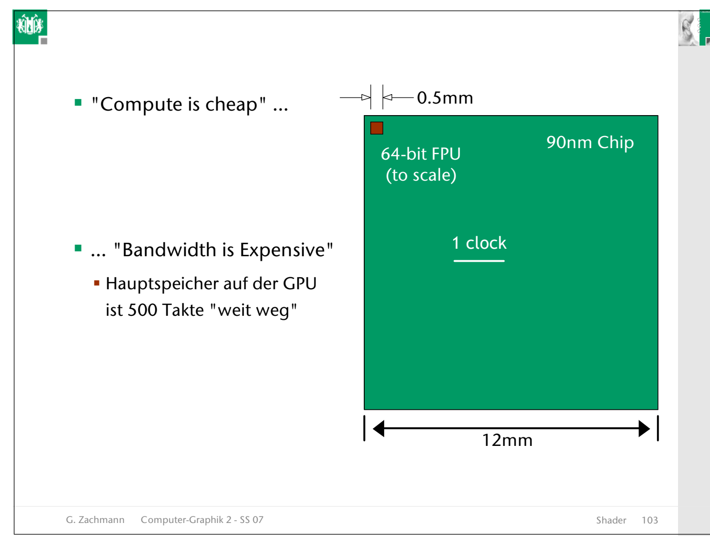


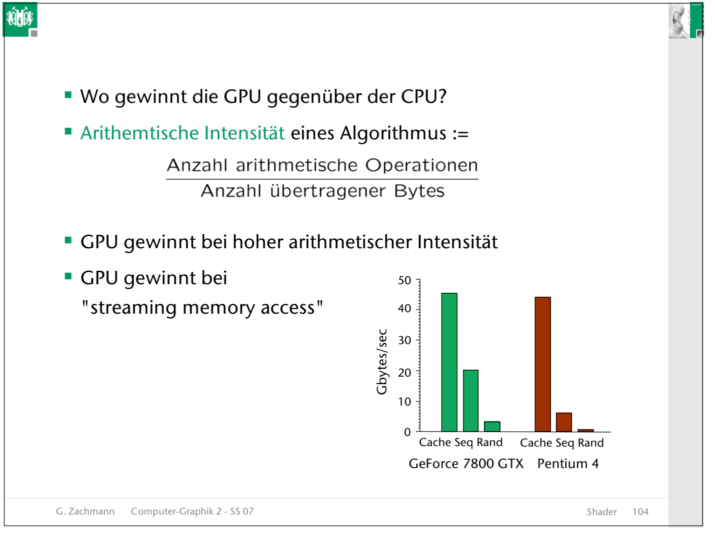
Tip zum Debugging

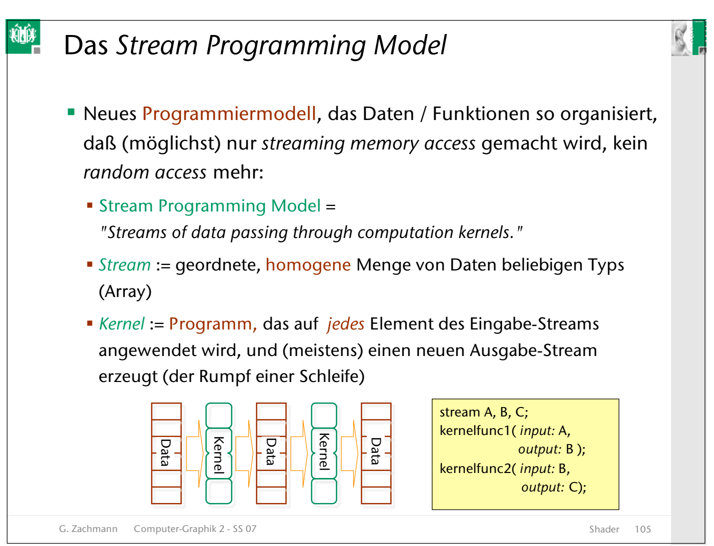
- Bei GPGPU-Anwendungen, nach jedem Pass die aktuelle "Read"-Textur anzeigen lassen
- Als Beispiel, wie man das macht: siehe die Funktion **`showReadTexture()`** in **`mandelbrot/mandelbrot`**.cpp im Zip-Archiv der Beispiele auf der Homepage der VL

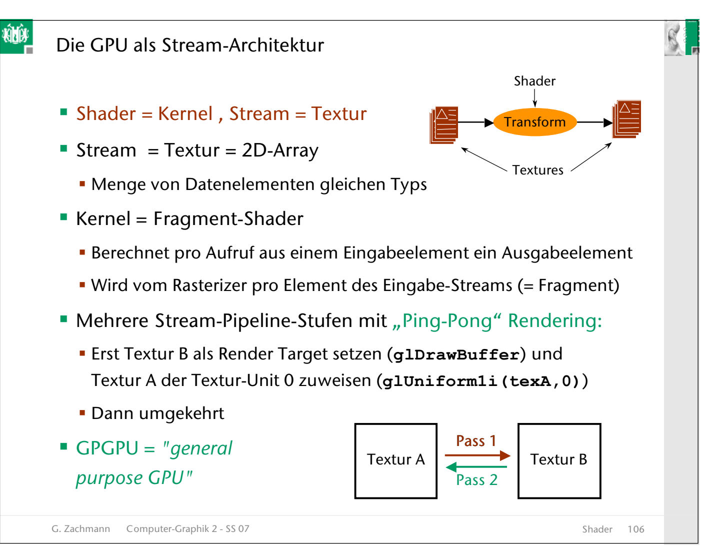
Aktueller Trend bzgl. Rechenleistung

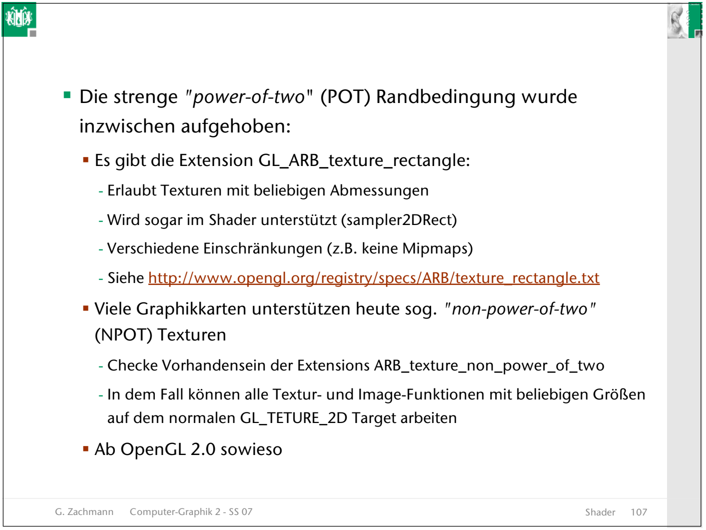



"Compute is cheap" ...
0.5mm
90nm Chip
64-bit FPU
(to scale)
1 clock
12mm
... "Bandwidth is Expensive"
Hauptspeicher auf der GPU ist 500 Takte "weit weg"
G. Zachmann Computer-Graphik 2 - SS 07
Shader 103


Wo gewinnt die GPU gegenüber der CPU?
Arithemtische Intensität eines Algorithmus :=
Anzahl arithmetische Operationen
Anzahl übertragener Bytes
GPU gewinnt bei hoher arithmetischer Intensität
GPU gewinnt bei "streaming memory access"
Gbytes/sec
0
10
20
30
40
50
Cache Seq Rand
Cache Seq Rand
GeForce 7800 GTX
Pentium 4
G. Zachmann Computer-Graphik 2 - SS 07
Shader 104


Das Stream Programming Model
Neues Programmiermodell, das Daten / Funktionen so organisiert, daß (möglichst) nur streaming memory access gemacht wird, kein random access mehr:
Stream Programming Model = "Streams of data passing through computation kernels."
Stream := geordnete, homogene Menge von Daten beliebigen Typs (Array)
Kernel := Programm, das auf jedes Element des Eingabe-Streams angewendet wird, und (meistens) einen neuen Ausgabe-Stream erzeugt (der Rumpf einer Schleife)
Data
Kernel
Data
Kernel
Data
stream A, B, C;
kernelfunc1(input: A,
output: B);
kernelfunc2(input: B,
output: C);
G. Zachmann Computer-Graphik 2 - SS 07
Shader 105


Die GPU als Stream-Architektur
Shader = Kernel , Stream = Textur
Stream = Textur = 2D-Array
Menge von Datenelementen gleichen Typs
Kernel = Fragment-Shader
Berechnet pro Aufruf aus einem Eingabeelement ein Ausgabeelement
Wird vom Rasterizer pro Element des Eingabe-Streams (= Fragment)
Mehrere Stream-Pipeline-Stufen mit „Ping-Pong" Rendering:
Erst Textur B als Render Target setzen (glDrawBuffer) und Textur A der Textur-Unit 0 zuweisen (glUniform1i(texA,0))
Dann umgekehrt
GPGPU = "general purpose GPU"
Shader
Transform
Textures
Textur A
Pass 1
Pass 2
Textur B
G. Zachmann Computer-Graphik 2 - SS 07
Shader 106



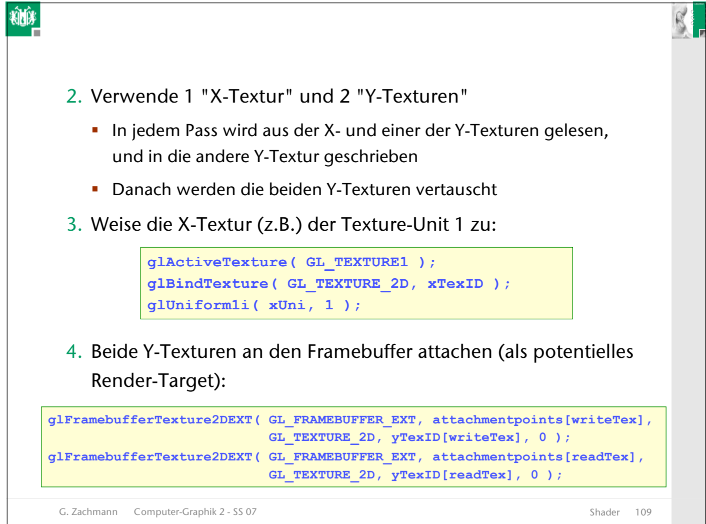
Weiteres Beispiel für GPGPU

- Die "saxpy"-Operation:
  - Steht für *"scalar alpha x plus y"*: $\mathbf{r} = \alpha\mathbf{x} + \mathbf{y}$
    wobei **x** und **y** Vektoren sind (gibt noch "daxpy", "caxpy", ...)
  - Ist eine der elementarsten Operationen in vielen Linear-Algebra-SW
- Beispiel: "saxpy" n-Mal ausführen

$$\mathbf{y}_{i+1} = \alpha\mathbf{x} + \mathbf{y}_i$$

1. Schritt: Vektoren **x** und **y** in 2D-Texturen übertragen
   - Packe dabei die ersten 4 Elemente der Vektoren in die 4 Kanäle des ersten Texels, ...

```
glBindTexture( GL_TEXTURE_2D, texID);
glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, texSizeX, texSizeY,
                 GL_RGBA, GL_FLOAT, data );
```

5. Die Rollen der beiden Y-Texturen "richtig" festlegen:

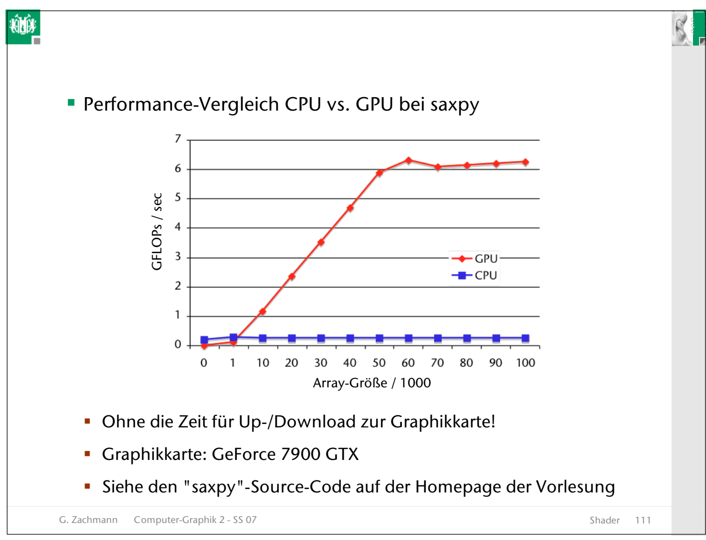
```
glDrawBuffer( attachmentpoints[writeTex] );
glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, yTexID[readTex] );
glUniform1i( yUni, 0 );                    // texunit 0
```

6. Shader ausführen (screen-sized quad rendern):

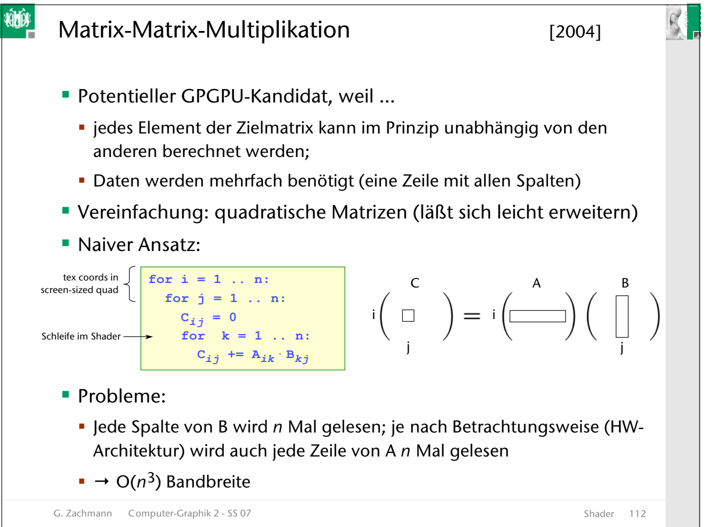
```
glBegin(GL_QUADS);
  glTexCoord2f( 0.0, 0.0 ); glVertex2f( 0.0,      0.0 );
  glTexCoord2f( 1.0, 0.0 ); glVertex2f( texSizeX, 0.0 ); ...
```

7. Die Rollen der beiden Y-Texturen vertauschen:

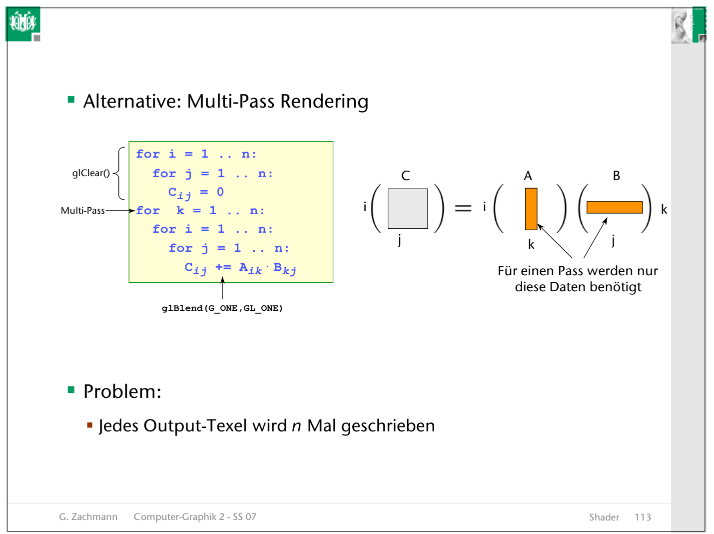
```
int h = writeTex; writeTex = readTex; readTex = h;
glDrawBuffer ...
```



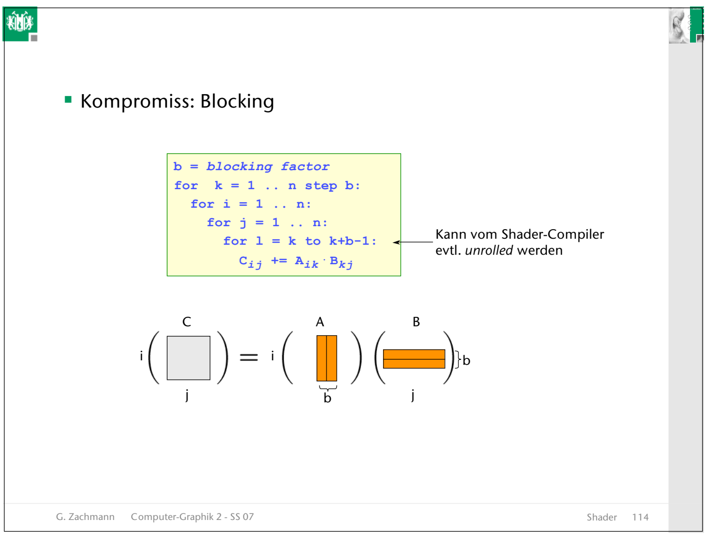
Performance-Vergleich CPU vs. GPU bei saxpy
GFLOPs / sec
Array-Größe / 1000
GPU
CPU
Ohne die Zeit für Up-/Download zur Graphikkarte!
Graphikkarte: GeForce 7900 GTX
Siehe den "saxpy"-Source-Code auf der Homepage der Vorlesung
G. Zachmann Computer-Graphik 2 - SS 07 Shader 111



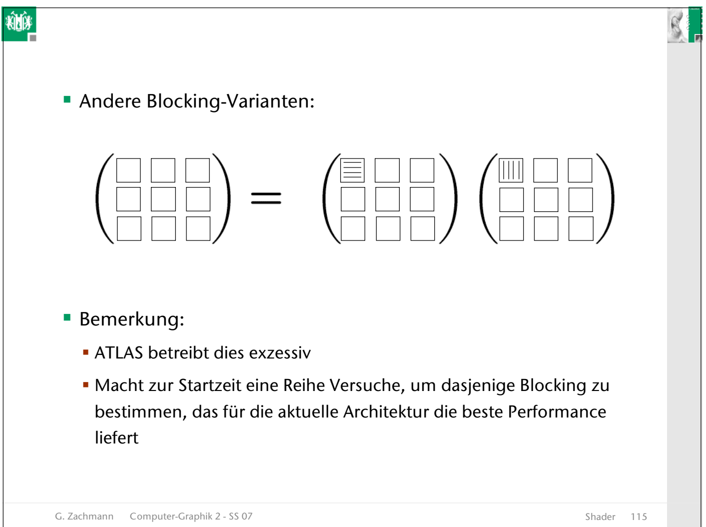
Matrix-Matrix-Multiplikation [2004]
Potentieller GPGPU-Kandidat, weil ...
jedes Element der Zielmatrix kann im Prinzip unabhängig von den anderen berechnet werden;
Daten werden mehrfach benötigt (eine Zeile mit allen Spalten)
Vereinfachung: quadratische Matrizen (läßt sich leicht erweitern)
Naiver Ansatz:
tex coords in screen-sized quad
Schleife im Shader
for i = 1 .. n:
for j = 1 .. n:
Cij = 0
for k = 1 .. n:
Cij += Aik · Bkj
Probleme:
Jede Spalte von B wird n Mal gelesen; je nach Betrachtungsweise (HW-Architektur) wird auch jede Zeile von A n Mal gelesen
→ O(n3) Bandbreite
G. Zachmann Computer-Graphik 2 - SS 07 Shader 112

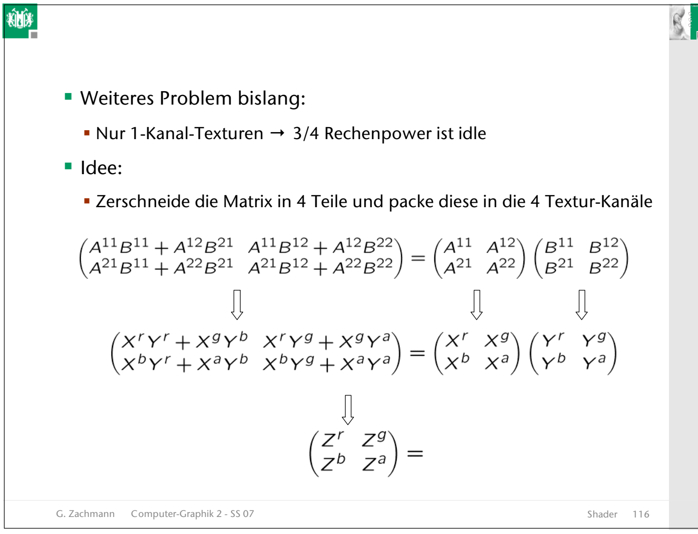


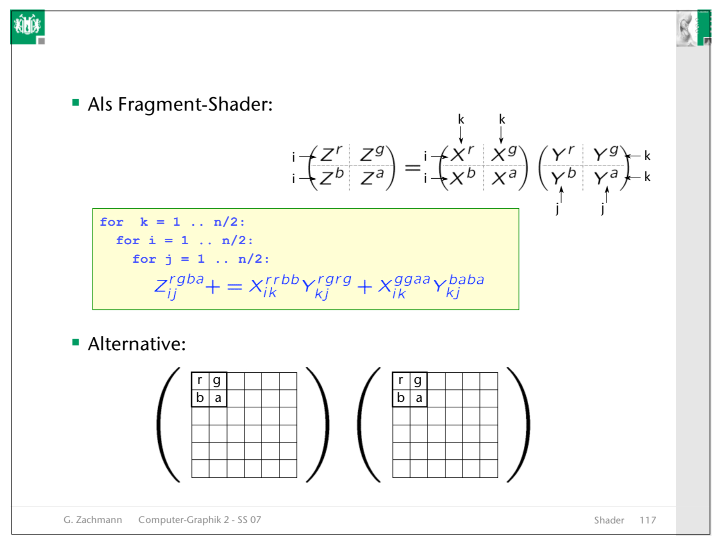
Alternative: Multi-Pass Rendering
glClear()
Multi-Pass
for i = 1 .. n:
for j = 1 .. n:
Cij = 0
for k = 1 .. n:
for i = 1 .. n:
for j = 1 .. n:
Cij += Aik · Bkj
glBlend(G_ONE,GL_ONE)
Für einen Pass werden nur diese Daten benötigt
Problem:
Jedes Output-Texel wird n Mal geschrieben
G. Zachmann Computer-Graphik 2 - SS 07 Shader 113

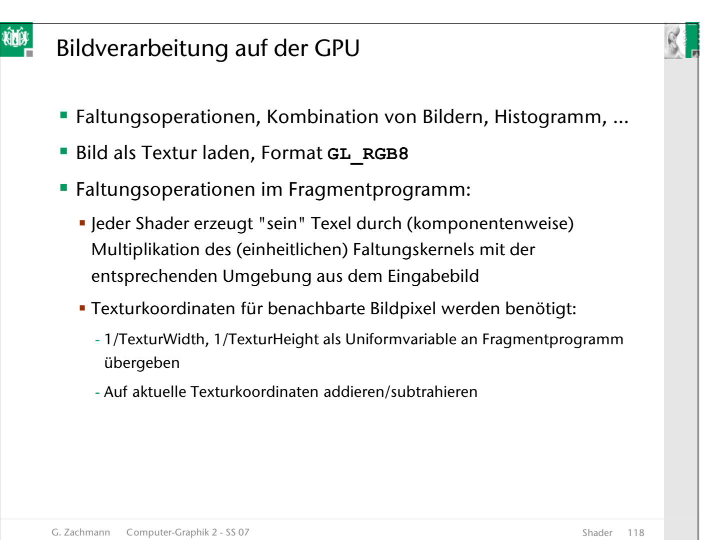


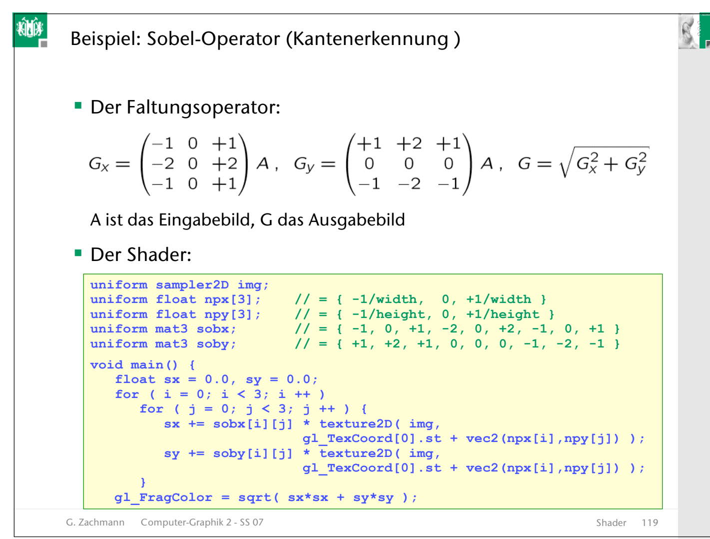
Kompromiss: Blocking
b = blocking factor
for k = 1 .. n step b:
for i = 1 .. n:
for j = 1 .. n:
for l = k to k+b-1:
Cij += Aik · Bkj
Kann vom Shader-Compiler evtl. unrolled werden
G. Zachmann Computer-Graphik 2 - SS 07 Shader 114

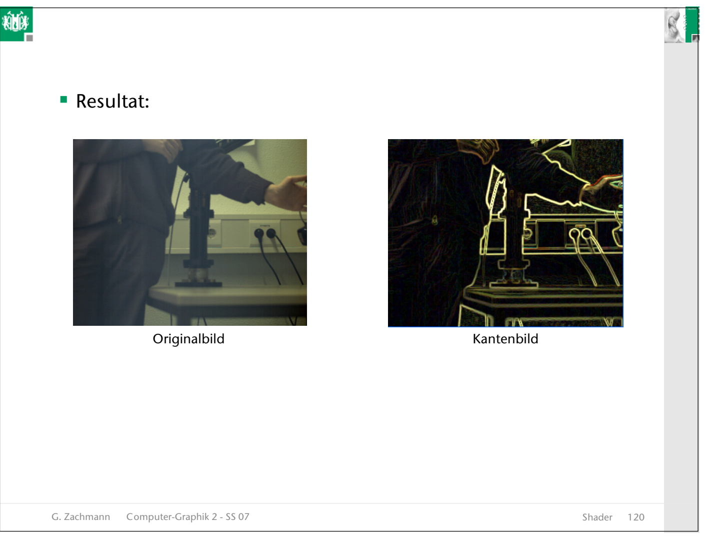

Andere Blocking-Varianten:
Bemerkung:
ATLAS betreibt dies exzessiv
Macht zur Startzeit eine Reihe Versuche, um dasjenige Blocking zu bestimmen, das für die aktuelle Architektur die beste Performance liefert
G. Zachmann Computer-Graphik 2 - SS 07
Shader 115

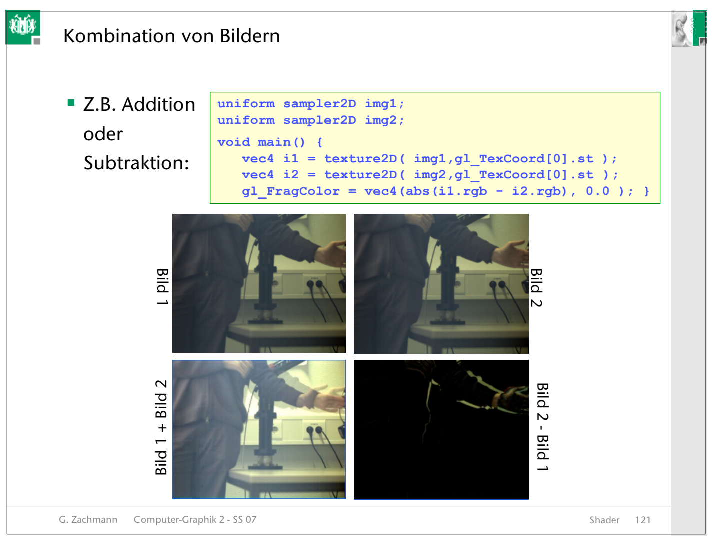

Weiteres Problem bislang:
Nur 1-Kanal-Texturen → 3/4 Rechenpower ist idle
Idee:
Zerschneide die Matrix in 4 Teile und packe diese in die 4 Textur-Kanäle
G. Zachmann Computer-Graphik 2 - SS 07
Shader 116

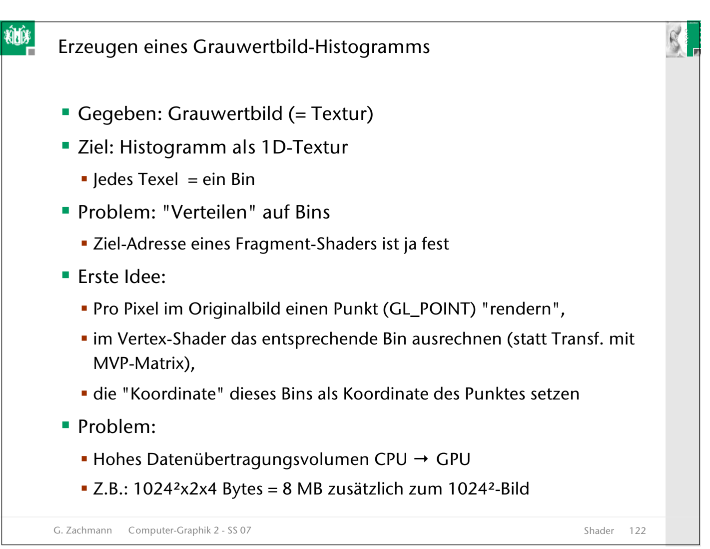

Als Fragment-Shader:
for k = 1 .. n/2:
for i = 1 .. n/2:
for j = 1 .. n/2:
Alternative:
r g
b a
r g
b a
G. Zachmann Computer-Graphik 2 - SS 07
Shader 117

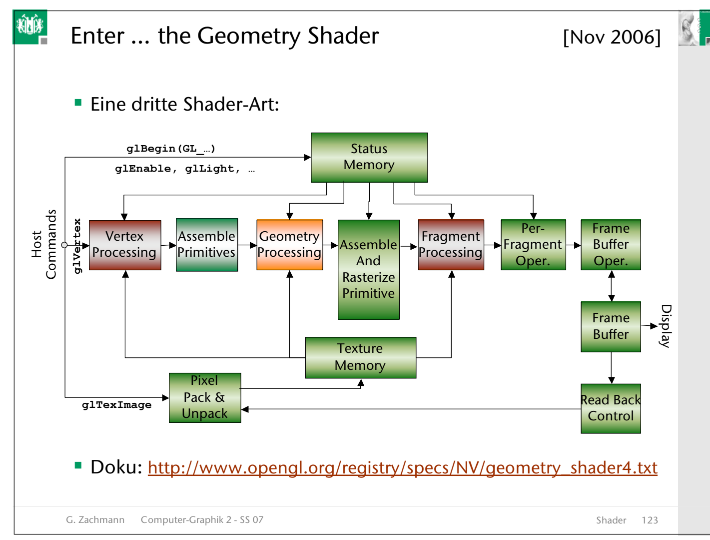

Bildverarbeitung auf der GPU
Faltungsoperationen, Kombination von Bildern, Histogramm, ...
Bild als Textur laden, Format GL_RGB8
Faltungsoperationen im Fragmentprogramm:
Jeder Shader erzeugt "sein" Texel durch (komponentenweise) Multiplikation des (einheitlichen) Faltungskernels mit der entsprechenden Umgebung aus dem Eingabebild
Texturkoordinaten für benachbarte Bildpixel werden benötigt:
1/TexturWidth, 1/TexturHeight als Uniformvariable an Fragmentprogramm übergeben
Auf aktuelle Texturkoordinaten addieren/subtrahieren
G. Zachmann Computer-Graphik 2 - SS 07
Shader 118

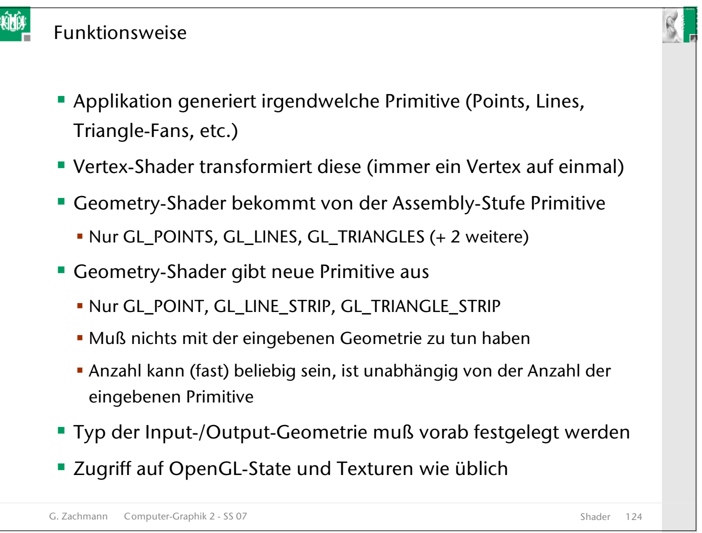

Beispiel: Sobel-Operator (Kantenerkennung)
Der Faltungsoperator:
$G_x = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix} A\,, \quad G_y = \begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} A\,, \quad G = \sqrt{G_x^2 + G_y^2}$
A ist das Eingabebild, G das Ausgabebild
Der Shader:
uniform sampler2D img;
uniform float npx[3]; // = { -1/width, 0, +1/width }
uniform float npy[3]; // = { -1/height, 0, +1/height }
uniform mat3 sobx; // = { -1, 0, +1, -2, 0, +2, -1, 0, +1 }
uniform mat3 soby; // = { +1, +2, +1, 0, 0, 0, -1, -2, -1 }
void main() {
float sx = 0.0, sy = 0.0;
for (i = 0; i < 3; i ++)
for (j = 0; j < 3; j ++) {
sx += sobx[i][j] * texture2D(img,
gl_TexCoord[0].st + vec2(npx[i],npy[j]));
sy += soby[i][j] * texture2D(img,
gl_TexCoord[0].st + vec2(npx[i],npy[j]));
}
gl_FragColor = sqrt(sx*sx + sy*sy);
G. Zachmann Computer-Graphik 2 - SS 07
Shader 119

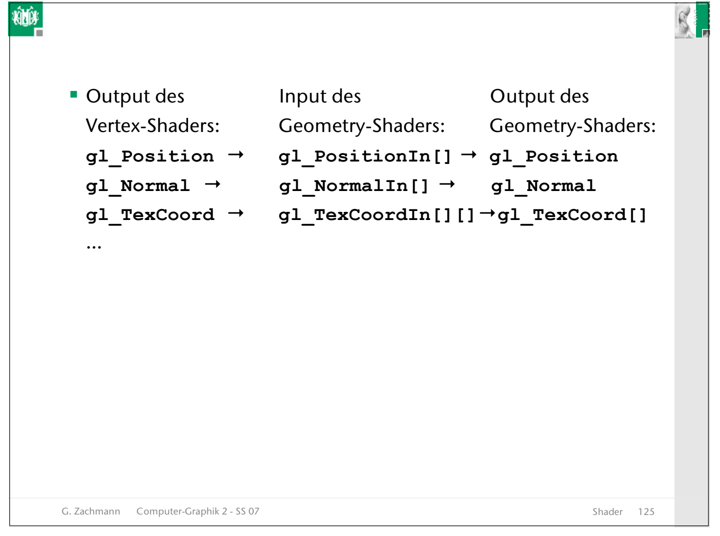

Resultat:
Originalbild
Kantenbild
G. Zachmann Computer-Graphik 2 - SS 07
Shader 120

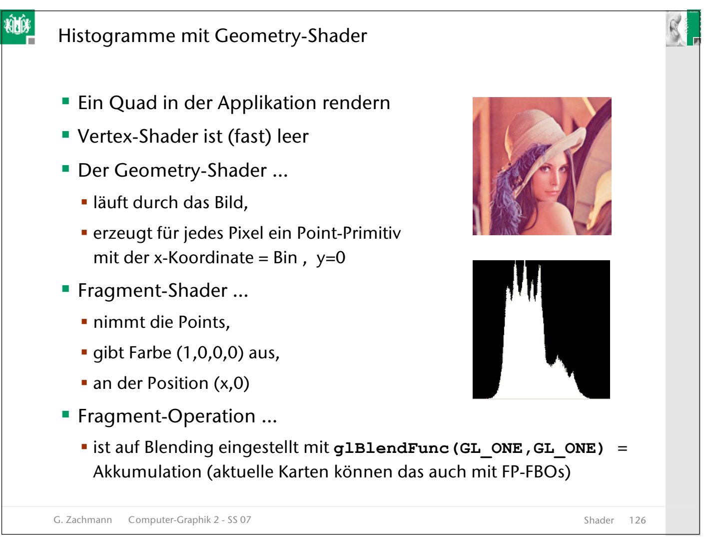

Kombination von Bildern
Z.B. Addition oder Subtraktion:
uniform sampler2D img1;
uniform sampler2D img2;
void main() {
vec4 i1 = texture2D(img1,gl_TexCoord[0].st);
vec4 i2 = texture2D(img2,gl_TexCoord[0].st);
gl_FragColor = vec4(abs(i1.rgb - i2.rgb), 0.0); }
Bild 1
Bild 2
Bild 1 + Bild 2
Bild 2 - Bild 1
G. Zachmann Computer-Graphik 2 - SS 07
Shader 121

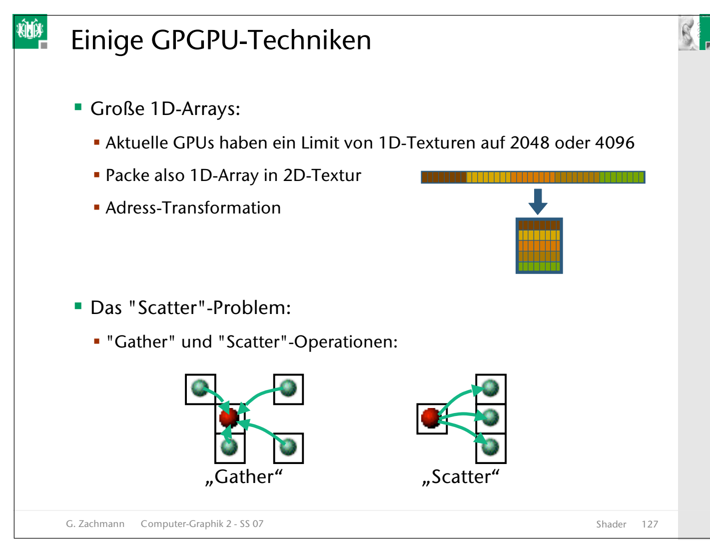

Erzeugen eines Grauwertbild-Histogramms
Gegeben: Grauwertbild (= Textur)
Ziel: Histogramm als 1D-Textur
Jedes Texel = ein Bin
Problem: "Verteilen" auf Bins
Ziel-Adresse eines Fragment-Shaders ist ja fest
Erste Idee:
Pro Pixel im Originalbild einen Punkt (GL_POINT) "rendern",
im Vertex-Shader das entsprechende Bin ausrechnen (statt Transf. mit MVP-Matrix),
die "Koordinate" dieses Bins als Koordinate des Punktes setzen
Problem:
Hohes Datenübertragungsvolumen CPU → GPU
Z.B.: 1024²x2x4 Bytes = 8 MB zusätzlich zum 1024²-Bild
G. Zachmann Computer-Graphik 2 - SS 07
Shader 122

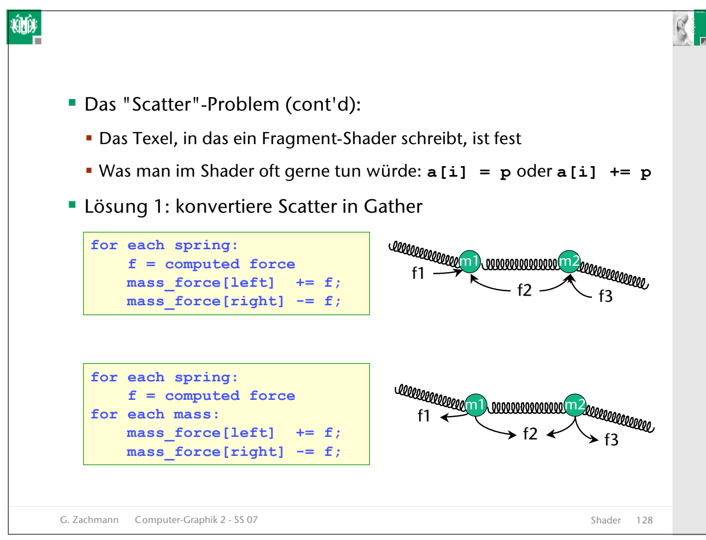

Enter ... the Geometry Shader
[Nov 2006]
Eine dritte Shader-Art:
glBegin(GL_…)
glEnable, glLight, …
Status Memory
Host Commands
glVertex
Vertex Processing
Assemble Primitives
Geometry Processing
Assemble And Rasterize Primitive
Fragment Processing
Per-Fragment Oper.
Frame Buffer Oper.
Frame Buffer
Display
Texture Memory
Pixel Pack & Unpack
glTexImage
Read Back Control
Doku: http://www.opengl.org/registry/specs/NV/geometry_shader4.txt
G. Zachmann Computer-Graphik 2 - SS 07
Shader 123

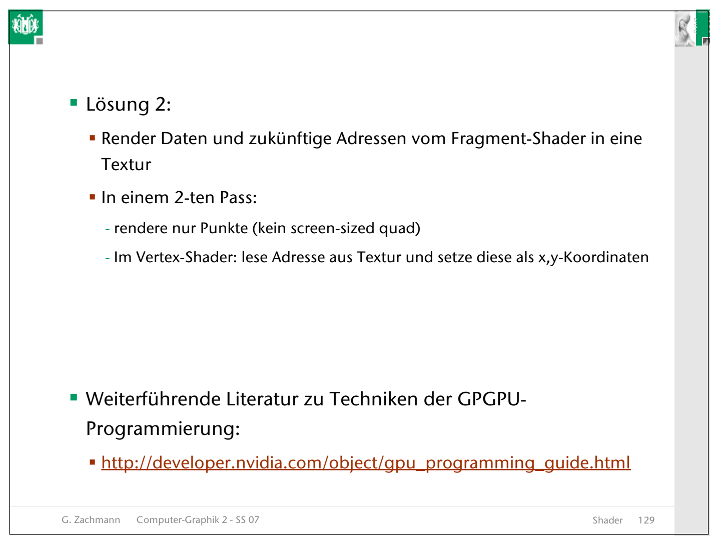

Funktionsweise
Applikation generiert irgendwelche Primitive (Points, Lines, Triangle-Fans, etc.)
Vertex-Shader transformiert diese (immer ein Vertex auf einmal)
Geometry-Shader bekommt von der Assembly-Stufe Primitive
Nur GL_POINTS, GL_LINES, GL_TRIANGLES (+ 2 weitere)
Geometry-Shader gibt neue Primitive aus
Nur GL_POINT, GL_LINE_STRIP, GL_TRIANGLE_STRIP
Muß nichts mit der eingebenen Geometrie zu tun haben
Anzahl kann (fast) beliebig sein, ist unabhängig von der Anzahl der eingebenen Primitive
Typ der Input-/Output-Geometrie muß vorab festgelegt werden
Zugriff auf OpenGL-State und Texturen wie üblich
G. Zachmann Computer-Graphik 2 - SS 07
Shader 124

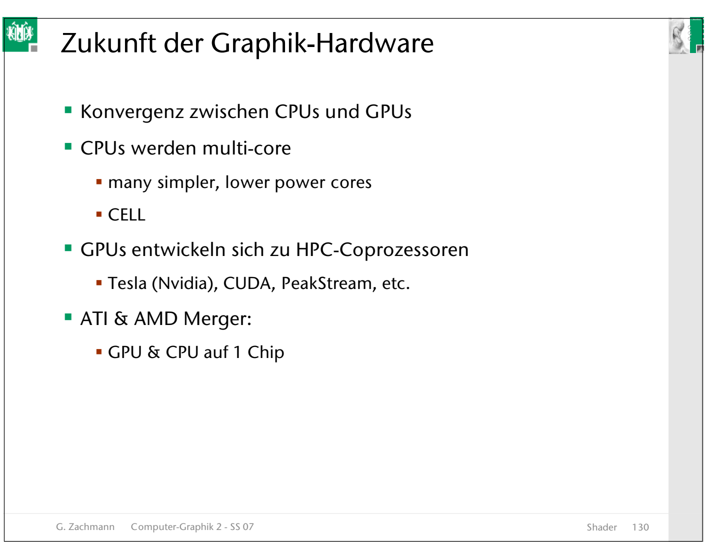

Output des Vertex-Shaders:
gl_Position →
gl_Normal →
gl_TexCoord →
…
Input des Geometry-Shaders:
gl_PositionIn[] →
gl_NormalIn[] →
gl_TexCoordIn[][] →
Output des Geometry-Shaders:
gl_Position
gl_Normal
gl_TexCoord[]
G. Zachmann Computer-Graphik 2 - SS 07
Shader 125


Histogramme mit Geometry-Shader
Ein Quad in der Applikation rendern
Vertex-Shader ist (fast) leer
Der Geometry-Shader ...
läuft durch das Bild,
erzeugt für jedes Pixel ein Point-Primitiv mit der x-Koordinate = Bin , y=0
Fragment-Shader ...
nimmt die Points,
gibt Farbe (1,0,0,0) aus,
an der Position (x,0)
Fragment-Operation ...
ist auf Blending eingestellt mit glBlendFunc(GL_ONE,GL_ONE) = Akkumulation (aktuelle Karten können das auch mit FP-FBOs)
G. Zachmann Computer-Graphik 2 - SS 07
Shader 126


Einige GPGPU-Techniken
Große 1D-Arrays:
Aktuelle GPUs haben ein Limit von 1D-Texturen auf 2048 oder 4096
Packe also 1D-Array in 2D-Textur
Adress-Transformation
Das "Scatter"-Problem:
"Gather" und "Scatter"-Operationen:
„Gather"
„Scatter"
G. Zachmann Computer-Graphik 2 - SS 07
Shader 127


Das "Scatter"-Problem (cont'd):
Das Texel, in das ein Fragment-Shader schreibt, ist fest
Was man im Shader oft gerne tun würde: a[i] = p oder a[i] += p
Lösung 1: konvertiere Scatter in Gather
for each spring:
f = computed force
mass_force[left] += f;
mass_force[right] -= f;
m1
m2
f1
f2
f3
for each spring:
f = computed force
for each mass:
mass_force[left] += f;
mass_force[right] -= f;
G. Zachmann Computer-Graphik 2 - SS 07
Shader 128


Lösung 2:
Render Daten und zukünftige Adressen vom Fragment-Shader in eine Textur
In einem 2-ten Pass:
- rendere nur Punkte (kein screen-sized quad)
- Im Vertex-Shader: lese Adresse aus Textur und setze diese als x,y-Koordinaten
Weiterführende Literatur zu Techniken der GPGPU-Programmierung:
http://developer.nvidia.com/object/gpu_programming_guide.html
G. Zachmann Computer-Graphik 2 - SS 07
Shader 129


Zukunft der Graphik-Hardware
Konvergenz zwischen CPUs und GPUs
CPUs werden multi-core
many simpler, lower power cores
CELL
GPUs entwickeln sich zu HPC-Coprozessoren
Tesla (Nvidia), CUDA, PeakStream, etc.
ATI & AMD Merger:
GPU & CPU auf 1 Chip
G. Zachmann Computer-Graphik 2 - SS 07
Shader 130