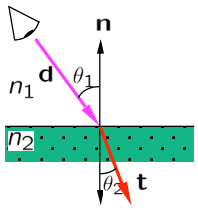
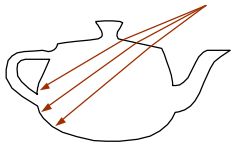
Lichtbrechung

- 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

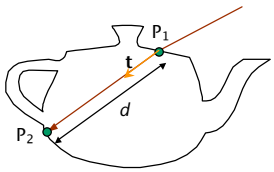


- Warum also ist Brechung so schwer?
  - Um den korrekten Schnittpunkt des gebrochenen Strahls zu berechnen, benötigt man die gesamte Geometrie!

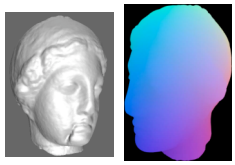


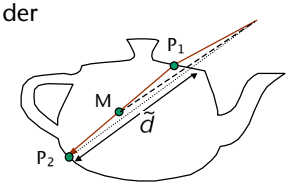
- Ziel: transparente Objekte mit 2 Schnittflächen approximieren
- Schritt 1: bestimme den nächsten Schnittpunkt

$$P_2 = P_1 + d\mathbf{t}$$

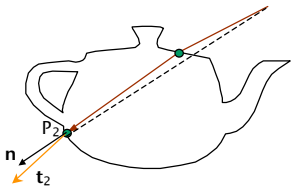


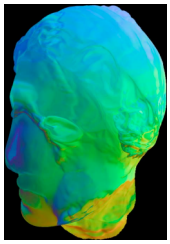
  - Idee: approximiere *d*
  - Rendere dazu 1x in einem Pass vorab eine Depth-Map der backfacing Polygone vom Viewpoint aus
  - Suche mit Binärsuche (ca. 5 Iter.) darin nach der "richtigen" Tiefe:
    - Bestimme Midpoint
    - Projiziere Midpoint bzgl. Viewpoint nach 2D
    - Indiziere damit die Depth





- 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





Normal-Map

- Viele Probleme:
  - Bei *depth complexity* > 2:
    - Welche Normale / welcher Tiefenwert soll behalten werden?
  - Approximation der Distanz
  - Aliasing

Beispiele

Mit innerer Reflexion

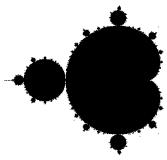
GPGPU

- Die Mandelbrot-Menge:
  - Bilde zu jedem $c \in \mathbb{C}$ die (unendliche) Folge

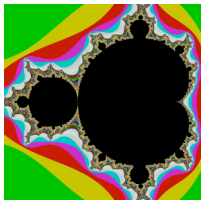
$$z_{i+1} = z_i^2 + c\,, \quad z_0 = 0$$

  - Definiere die Mandelbrot-Menge

$$\mathbb{M} = \{c \in \mathbb{C} \mid \text{Folge } (z_i) \text{ bleibt beschränkt } \}$$

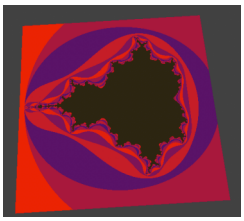
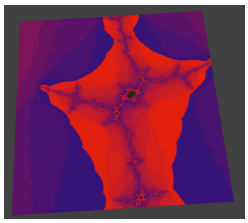
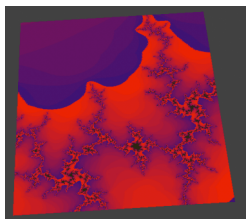
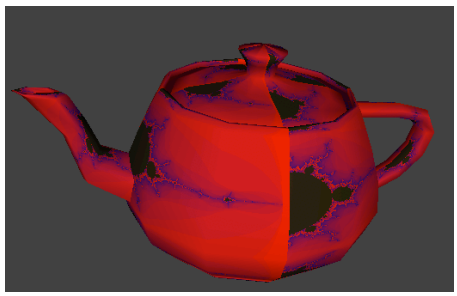


- Satz (o. Bew.):
  $\exists\, t : |z_t| > 2 \;\Rightarrow\; c$ ist nicht in der Mandelbrotmenge
- Hübsche Visualisierung der Mandelbrotmenge:
  - Färbe Pixel $c = (x,y)$ schwarz falls $|z|$ nach "vielen" Iterationen immer noch < 2
  - Färbe c abhängig von der Anzahl Iterationen t, die nötig waren, bis $|z_t| > 2$ wurde



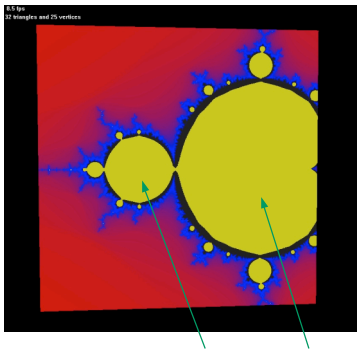
- Einige interessante Fakten zur Mandelbrot-Menge (mit denen man bei Partys beeindrucken kann ;-)):
  - Die Länge des Randes der Mandelbrot-Menge ist unendlich
  - Die Mandelbrot-Menge ist zusammenhängend (d.h., alle "schwarzen" Gebiete sind miteinander verbunden)
  - Es gibt zu jeder Farbe genau 1 Band 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"

Der Mandelbrot-Shader

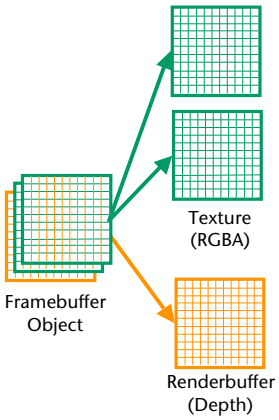
mandelbrot1.{vert,frag}
in
demos/shader/GLSL_editor
(s.a. Tar-File auf der Homepage der Vorlesung)

- Eine "Optimierung":
  - Eine beschränkte Folge von z_i kann gegen einen einzelnen (komplexen) Wert konvergieren,
  - oder gegen einen Zyklus von Werten,
  - oder chaotisch sein
- Idee:
  - Versuche, solche Zyklen zu erkennen und dann aus der Iteration auszubrechen (was hoffentlich früher passiert)
  - Führe dazu ein Array mit den letzten *k* Folgegliedern
- Leider: 4x langsamer als die brute-force Variante!
- Demo: **`mandelbrot2.{frag,rfx}`** auf der VL-Homepage



Framebuffer objects (FBOs)

- FBO = "off-screen frame buffer":
  - Abstraktes Speichermodell eines FBO = "struct of pointers to textures (and z buffers)"
  - Bindet GPU-Speicher an FBO als write-only
  - Kann floating-point-Werte speichern (also 4x32 Bit, statt 4x8 Bit)
- Erlaubt das Rendern direkt in eine Textur
  - Oder sogar mehrere
- Ersetzt ältere Techniken:
  - pbuffer, "uberbuffer", superbuffer
  - Render-to-Texture



FBOs mit OpenGL

1.
```
GLuint fbo;
glGenFramebuffersEXT( 1, &fbo );
```

2.
```
glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbo );
```

3.
```
GLuint texID[2];
glGenTextures( 2, texID );
glBindTexture( GL_TEXTURE_2D, texID[0] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA32F_ARB,
         texSizeX, texSizeY , 0, GL_RGBA, GL_FLOAT, NULL );
```

4.
```
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
          GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D,
          texID[0], 0);
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
          GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D,
          texID[1], 0);
```


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

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

Anwendung auf die Mandelbrot-Menge

- Berechne pro Rendering-Schleifendurchlauf eine Iteration der Funktion, für alle Pixel (= Texel) → Multipass-Rendering
  - Speichere dazu pro Texel
    - z_i
    - Anzahl Iterationen, bei der die entsprechende z_i-Folge den Radius verließ
- Zerlege die Schleife in mehrere Phasen:
  1. Phase: initialisiere alle Texel mit dem entsprechenden $c = z_1$ im
  2. Phase: führe n Schleifendurchläufe durch, wobei in jedem Durchlauf der Fragment-Shader für jedes Texel $z_{i+1} = z_i^2 + c$ berechnet
  3. Phase: berechne aus der pro Texel gespeicherten Anzahl Iterationen bis zum "Verlassen des Radius" eine Farbe

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 );
}
```

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:

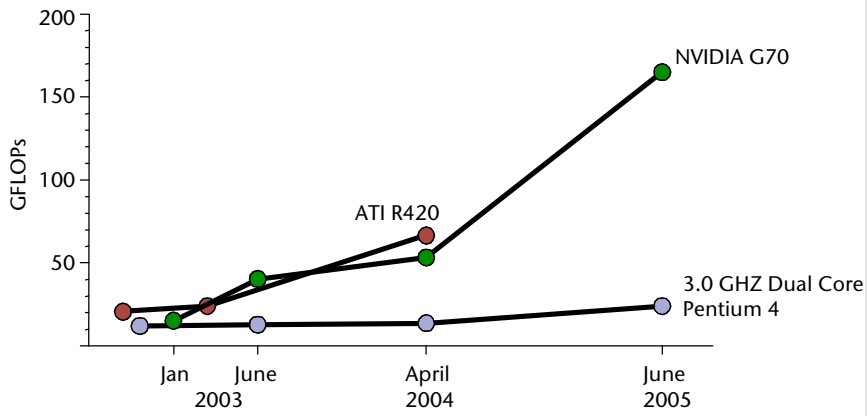
```
uniform sampler2D input;       // from last iteration
uniform float maxIterations;
void main ()
{
    // Lookup value from last iteration
    vec4 inputValue = texture2D(input, gl_TexCoord[0].xy);
    vec2 z = inputValue.xy;
    float n = inputValue.z;
    if ( n < maxIterations )
        // compute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

- Demo-Code: siehe Homepage der Vorlesung!

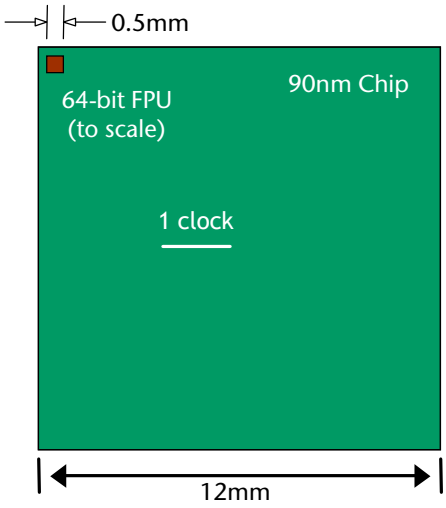
Tip zum Debugging

- 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



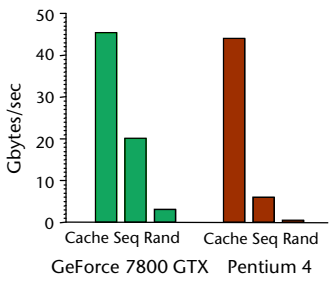
- "Compute is cheap" ...
- ... "Bandwidth is Expensive"
  - Hauptspeicher auf der GPU ist 500 Takte "weit weg"



- Wo gewinnt die GPU gegenüber der CPU?
- Arithemtische Intensität eines Algorithmus :=

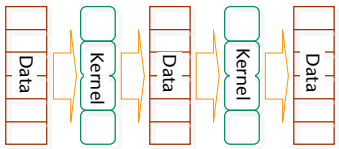
$$\frac{\text{Anzahl arithmetische Operationen}}{\text{Anzahl übertragener Bytes}}$$

- GPU gewinnt bei hoher arithmetischer Intensität
- GPU gewinnt bei "streaming memory access"



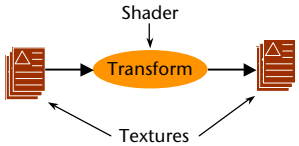
Das *Stream Programming Model*

- 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)

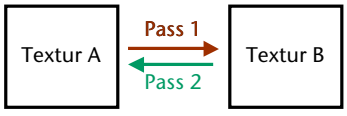



```
stream A, B, C;
kernelfunc1( input: A,
             output: B );
kernelfunc2( input: B,
             output: C);
```


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"*



- Die strenge *"power-of-two"* (POT) Randbedingung wurde inzwischen aufgehoben:
  - Es gibt die Extension GL_ARB_texture_rectangle:
    - Erlaubt Texturen mit beliebigen Abmessungen
    - Wird sogar im Shader unterstützt (sampler2DRect)
    - Verschiedene Einschränkungen (z.B. keine Mipmaps)
    - Siehe http://www.opengl.org/registry/specs/ARB/texture_rectangle.txt
  - Viele Graphikkarten unterstützen heute sog. *"non-power-of-two"* (NPOT) Texturen
    - Checke Vorhandensein der Extensions ARB_texture_non_power_of_two
    - In dem Fall können alle Textur- und Image-Funktionen mit beliebigen Größen auf dem normalen GL_TETURE_2D Target arbeiten
  - Ab OpenGL 2.0 sowieso

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 );
```

2. Verwende 1 "X-Textur" und 2 "Y-Texturen"
   - In jedem Pass wird aus der X- und einer der Y-Texturen gelesen, und in die andere Y-Textur geschrieben
   - Danach werden die beiden Y-Texturen vertauscht
3. Weise die X-Textur (z.B.) der Texture-Unit 1 zu:

```
glActiveTexture( GL_TEXTURE1 );
glBindTexture( GL_TEXTURE_2D, xTexID );
glUniform1i( xUni, 1 );
```

4. Beide Y-Texturen an den Framebuffer attachen (als potentielles Render-Target):

```
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, attachmentpoints[writeTex],
                           GL_TEXTURE_2D, yTexID[writeTex], 0 );
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, attachmentpoints[readTex],
                           GL_TEXTURE_2D, yTexID[readTex], 0 );
```


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
```

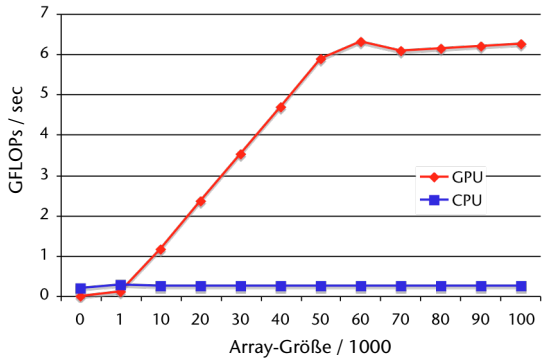
6. Shader ausführen (screen-sized quad rendern):

```
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



  - Ohne die Zeit für Up-/Download zur Graphikkarte!
  - Graphikkarte: GeForce 7900 GTX
  - Siehe den "saxpy"-Source-Code auf der Homepage der Vorlesung

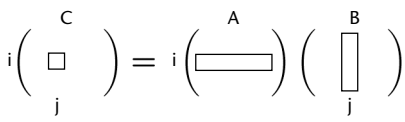
Matrix-Matrix-Multiplikation [2004]

- 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

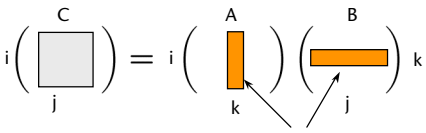
```
for i = 1 .. n:
  for j = 1 .. n:
    C_ij = 0
    for  k = 1 .. n:
      C_ij += A_ik · B_kj
```



- Probleme:
  - Jede Spalte von B wird *n* Mal gelesen; je nach Betrachtungsweise (HW-Architektur) wird auch jede Zeile von A *n* Mal gelesen
  - → $O(n^3)$ Bandbreite

- Alternative: Multi-Pass Rendering

```
glClear()  for i = 1 .. n:
             for j = 1 .. n:
               Cij = 0
Multi-Pass → for  k = 1 .. n:
             for i = 1 .. n:
               for j = 1 .. n:
                 Cij += Aik · Bkj
                          ↑
                 glBlend(G_ONE,GL_ONE)
```



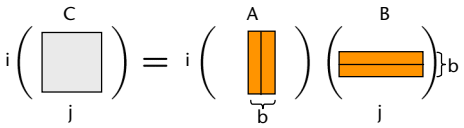
Für einen Pass werden nur diese Daten benötigt

- Problem:
  - Jedes Output-Texel wird *n* Mal geschrieben

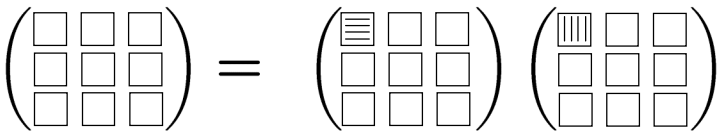
- Kompromiss: Blocking

```
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



- 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

- 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

$$\begin{pmatrix} A^{11}B^{11}+A^{12}B^{21} & A^{11}B^{12}+A^{12}B^{22} \\ A^{21}B^{11}+A^{22}B^{21} & A^{21}B^{12}+A^{22}B^{22} \end{pmatrix} = \begin{pmatrix} A^{11} & A^{12} \\ A^{21} & A^{22} \end{pmatrix} \begin{pmatrix} B^{11} & B^{12} \\ B^{21} & B^{22} \end{pmatrix}$$

$$\Downarrow \qquad\qquad \Downarrow \qquad \Downarrow$$

$$\begin{pmatrix} X^rY^r+X^gY^b & X^rY^g+X^gY^a \\ X^bY^r+X^aY^b & X^bY^g+X^aY^a \end{pmatrix} = \begin{pmatrix} X^r & X^g \\ X^b & X^a \end{pmatrix} \begin{pmatrix} Y^r & Y^g \\ Y^b & Y^a \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} Z^r & Z^g \\ Z^b & Z^a \end{pmatrix} =$$

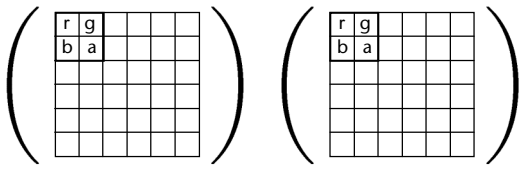
- Als Fragment-Shader:

$$\begin{pmatrix} Z^r & Z^g \\ Z^b & Z^a \end{pmatrix}_{i} = \begin{pmatrix} X^r & X^g \\ X^b & X^a \end{pmatrix}_{i,k} \begin{pmatrix} Y^r & Y^g \\ Y^b & Y^a \end{pmatrix}_{k,j}$$

```
for  k = 1 .. n/2:
  for i = 1 .. n/2:
    for j = 1 .. n/2:
```

$$Z_{ij}^{rgba} += X_{ik}^{rrbb} Y_{kj}^{rgrg} + X_{ik}^{ggaa} Y_{kj}^{baba}$$

- Alternative:



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

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

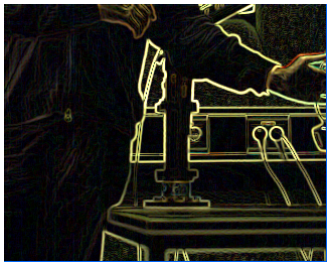
- Der Shader:

```
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 );
```


- Resultat:

Originalbild

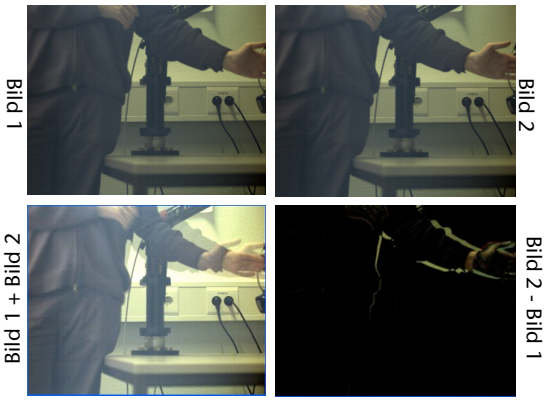


Kantenbild

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 ); }
```

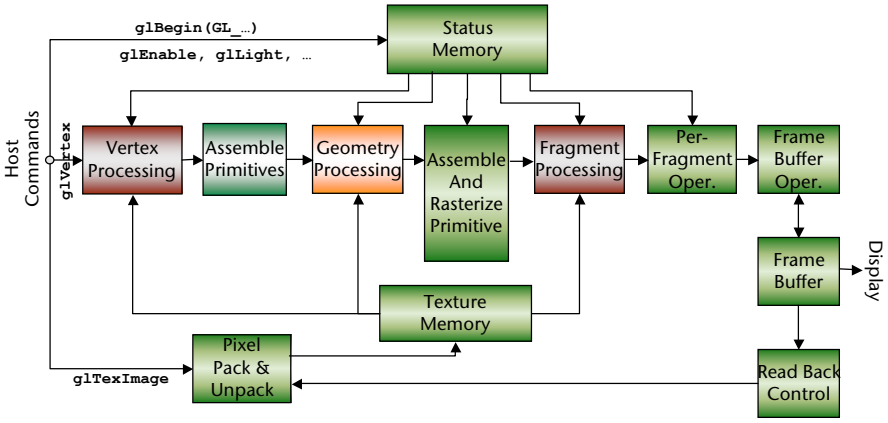


Erzeugen eines Grauwertbild-Histogramms

- 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.: 1024^2x2x4 Bytes = 8 MB zusätzlich zum 1024^2-Bild

Enter ... the Geometry Shader [Nov 2006]

- Eine dritte Shader-Art:



- Doku: http://www.opengl.org/registry/specs/NV/geometry_shader4.txt

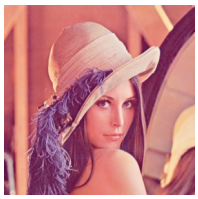
Funktionsweise

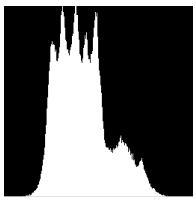
- Applikation generiert irgendwelche Primitive (Points, Lines, Triangle-Fans, etc.)
- Vertex-Shader transformiert diese (immer ein Vertex auf einmal)
- Geometry-Shader bekommt von der Assembly-Stufe Primitive
  - Nur GL_POINTS, GL_LINES, GL_TRIANGLES (+ 2 weitere)
- Geometry-Shader gibt neue Primitive 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

- 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[]`

Histogramme mit Geometry-Shader

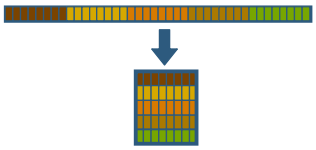
- Ein Quad in der Applikation rendern
- Vertex-Shader ist (fast) leer
- Der Geometry-Shader ...
  - läuft durch das Bild,
  - erzeugt für jedes Pixel ein Point-Primitiv mit der x-Koordinate = Bin , y=0
- Fragment-Shader ...
  - nimmt die Points,
  - gibt Farbe (1,0,0,0) aus,
  - an der Position (x,0)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



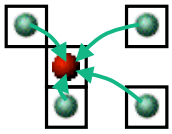


Einige GPGPU-Techniken

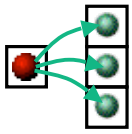
- Große 1D-Arrays:
  - Aktuelle GPUs haben ein Limit von 1D-Texturen auf 2048 oder 4096
  - Packe also 1D-Array in 2D-Textur
  - Adress-Transformation



- Das "Scatter"-Problem:
  - "Gather" und "Scatter"-Operationen:



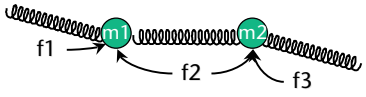
„Gather“



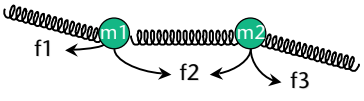
„Scatter“

- 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;
```




```
for each spring:
    f = computed force
for each mass:
    mass_force[left]  += f;
    mass_force[right] -= f;
```



- 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

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