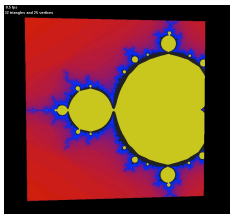
Computer-Graphik II
GPGPU-Algorithmen



G. Zachmann
Clausthal University, Germany
cg.in.tu-clausthal.de

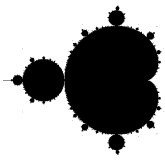
Parallele Berechnung der Mandelbrot-Menge

- 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

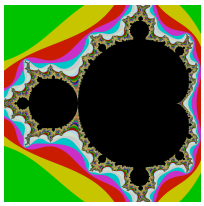
$$\mathbb{M} = \{ c \in \mathbb{C} \mid \text{Folge } (z_i) \text{ bleibt beschränkt } \}$$



- Satz (o. Bew.):

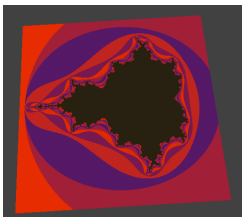
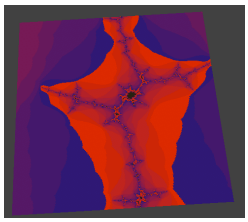
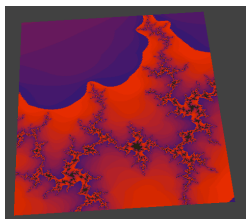
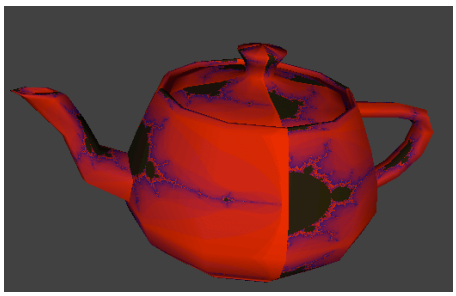
$$\exists\, t : |z_t| > 2 \ \Rightarrow\ c \text{ ist nicht in der Mandelbrotmenge}$$

- 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



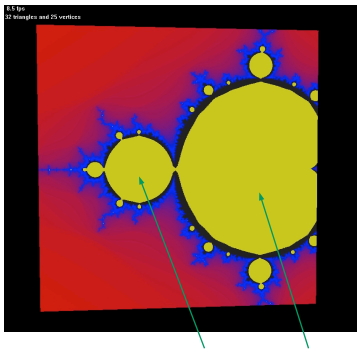
- Einige interessante Fakten zur Mandelbrot-Menge (mit denen man bei Partys beeindrucken kann ;-)):
  - Die Länge des Randes der Mandelbrot-Menge ist unendlich
  - Die Mandelbrot-Menge ist zusammenhängend (d.h., alle "schwarzen" Gebiete sind miteinander verbunden)
  - Es gibt zu jeder Farbe genau 1 Band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

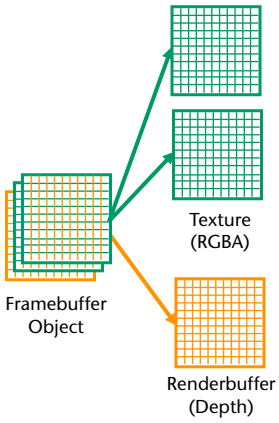
Demo in demos/shader/mandelbrot (s.a. Tar-File auf der Homepage der Vorlesung)

- Eine "Optimierung":
  - Eine beschränkte Folge von z_i kann entweder 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!



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-Pass 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:

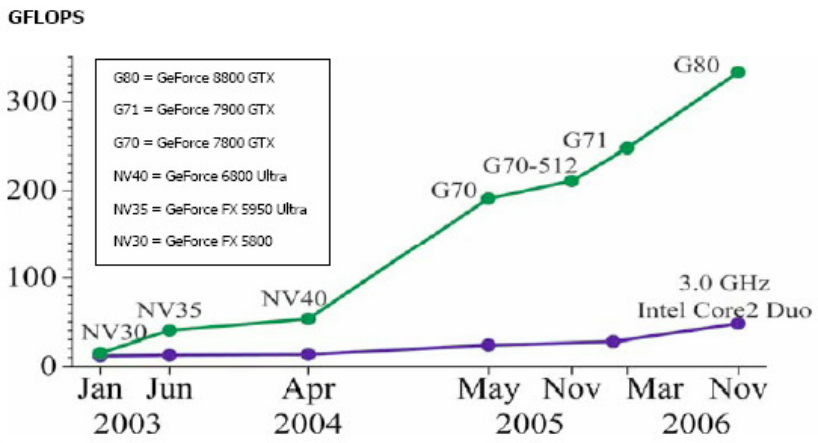
```
uniform sampler2D input;       // from last iteration
uniform float maxIterations;
void main ()
{
    // Lookup value from last iteration
    vec4 inputValue = texture2D(input, gl_TexCoord[0].xy);
    vec2 z = inputValue.xy;
    float n = inputValue.z;
    if ( n < maxIterations )
        // compute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

- Demo-Code: siehe Homepage der Vorlesung!

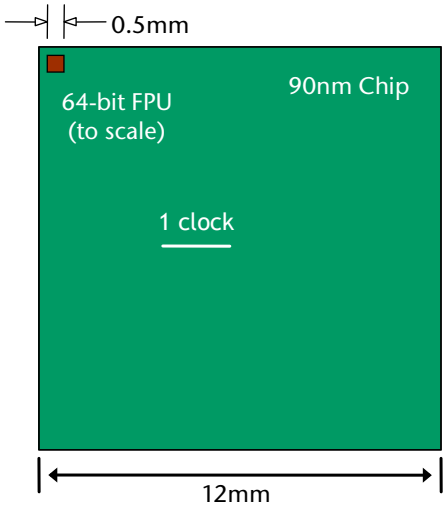
Tip zum Debugging

- 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



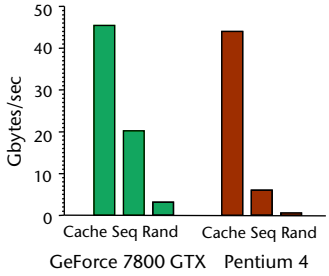
- "Compute is cheap" ...
- ... "Bandwidth is expensive"
  - Hauptspeicher (texture memory) ist 500 Takte "weit weg" von der GPU



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

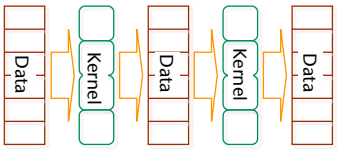
$$\frac{\text{Anzahl arithmetische Operationen}}{\text{Anzahl übertragener Bytes}}$$

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



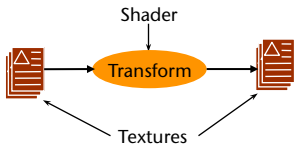
Das *Stream Programming Model*

- 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)

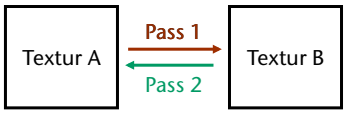



```
stream A, B, C;
kernelfunc1( input: A,
             output: B );
kernelfunc2( input: B,
             output: C);
```


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
  - Kostet meistens immer noch deutlich Performance

Lineare Algebra auf der GPU

- Am Beispiel der"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
4. Weise die X-Textur (z.B.) der Texture-Unit 1 zu:

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

5. 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
```

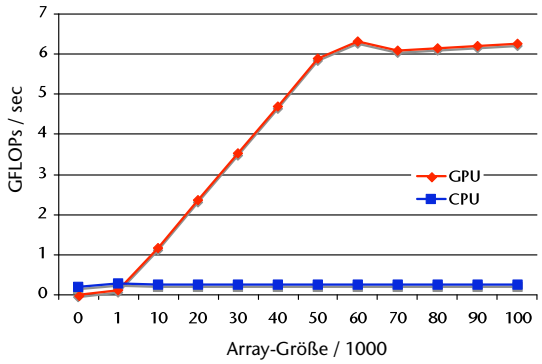
6. Shader ausführen (screen-sized quad rendern):

```
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

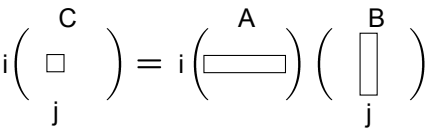
Matrix-Matrix-Multiplikation [2004]

- 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 →

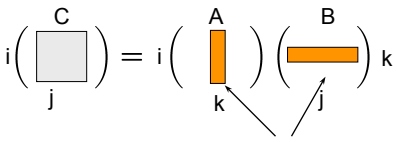
```
for i = 1 .. n:
  for j = 1 .. n:
    C_ij = 0
    for  k = 1 .. n:
      C_ij += A_ik · B_kj
```



- 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)
```



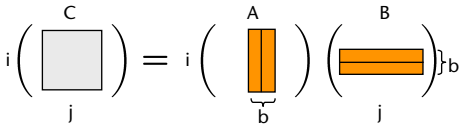
Für einen Pass werden nur diese Daten benötigt

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

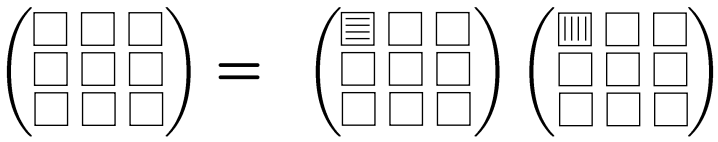
- Kompromiss: Blocking

```
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

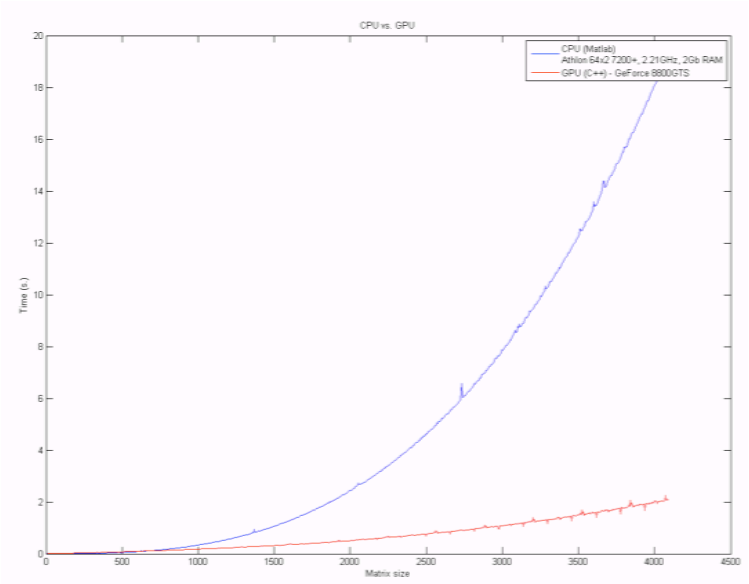


- Andere Blocking-Varianten:



- 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

Performance-Vergleich



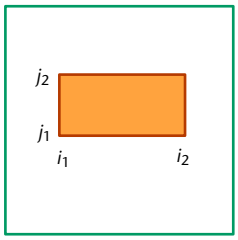
Dec 2007

Parallele Berechnung von Summed-Area Tables

- Gegeben: 2D-Array T mit Größe WxH
- Gesucht: eine Datenstruktur, so daß für beliebige i_1, i_2, j_1, j_2,

$$\sum_{k=i_1}^{i_2}\sum_{l=j_1}^{j_2} T(k,l)$$

in O(1) berechnet werden kann.



- Der Trick:

$$\sum_{k=i_1}^{i_2}\sum_{l=j_1}^{j_2} T(k,l) = \sum_{k=1}^{i_2}\sum_{l=1}^{j_2} T(k,l) - \sum_{k=1}^{i_1}\sum_{l=1}^{j_2} T(k,l) - \sum_{k=1}^{i_2}\sum_{l=1}^{j_1} T(k,l) + \sum_{k=1}^{i_1}\sum_{l=1}^{j_1} T(k,l)$$

- Definiere $S(i,j) = \sum_{k=1}^{i}\sum_{l=1}^{j} T(k,l)$

- Damit ist

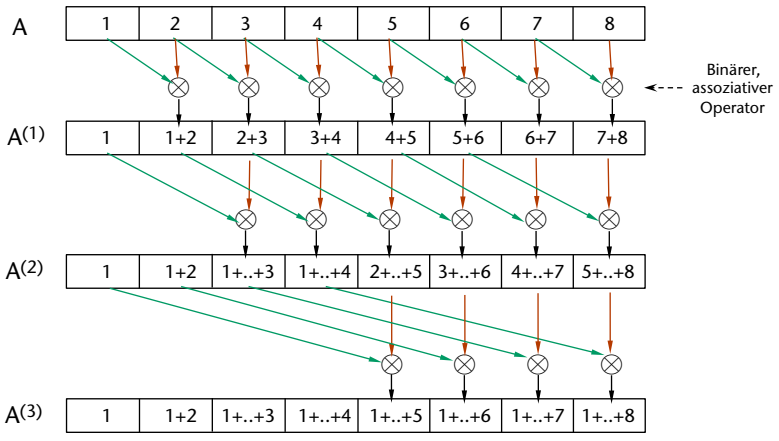
$$\sum_{k=i_1}^{i_2}\sum_{l=j_1}^{j_2} T(k,l) = S(i_2,j_2) - S(i_1,j_2) - S(i_2,j_1) + S(i_1,j_1)$$

- Aufgabe: aus T ein 2D-Array S berechnen, so daß

$$S(i,j) = \sum_{k=1}^{i} \sum_{l=1}^{j} T(k,l)$$

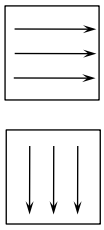
- Definition: S heißt "Summed-Area Table" (SAT)
- Parallele Berechnung einer SAT ist das 2D-Analogon zum sog. "Parallel Prefix Sum"-Problem

- Idee: verwende die Technik "recursive doubling"
- Beispiel in 1D:



Berechnung der Summed-Area Table auf der GPU

- 2 Phasen: horizontale Phase, vertikale Phase
  - Pro Phase $n = \lceil \log_2 W \rceil$ bzw. $m = \lceil \log_2 H \rceil$ Passes
- Annahme: pro Pass kann man 2 Texel lesen
- Horizontale Phase:



```
T = input array (texture)
S = output array
loop i = 1 .. n:
  S[s,t] = T[s,t] + T[s - 2^i, t]
  swap S, T
```

- Vertikale Phase: analog ...
- In der Praxis: mehr als 2 Texel pro Pass lesen!

- Problem: die Rechengenauigkeit!
- Annahme: T(*i*,*j*) benötigt b Bits
- Anzahl Bits für S:

$$\log_2(W \cdot H \cdot b) = \log_2 W + \log_2 H + b$$

- Beispiel:
  - Textur der Größe 256x256, jeder Eintrag mit 8 Bits
  - Anzahl Bits für S = 24!

Erhöhung der Rechengenauigkeit

1. Idee: "signed-offset" Repräsentation

- Annahme: alle $T(i,j) \in [0,1]$
- Setze

$$T'(i,j) = T(i,j) - t$$

wobei

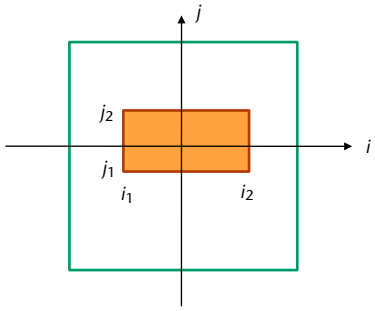
$$t = \text{ Mittelwert von T} = \frac{1}{wh}\sum_{1}^{w}\sum_{1}^{h} T(i,j)$$

- Dann ist

$$S'(i,j) = \sum_{1}^{i}\sum_{1}^{j} T'(i,j) = S(i,j) - i \cdot j \cdot t$$

- Bemerkung:
  - Man nutzt 1 Bit mehr aus (VZ-Bit)
  - Die SAT-Funktion S wächst nicht mehr monoton
  - $S'(w,h) \approx 0$
- Frage: wie berechnet man den Mittelwert auf der GPU?
  1. Idee: MIPMap von T berechnen, oberster Level der Pyramide = MW
  2. Idee: ohne MIPMap → Übungsaufgabe
     - Achtung: Algorithmus so entwerfen, daß man sich nicht wieder die Probleme mit der Rechengenauigkeit einfängt!

- Idee: verlege das ij-Koordinatensystem



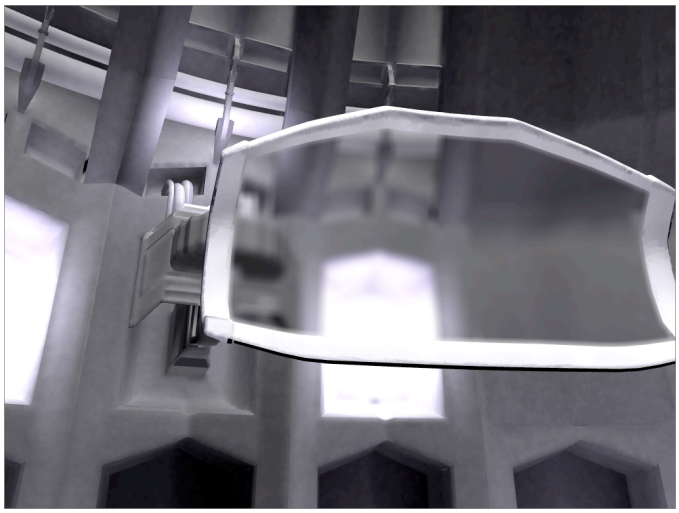
- Damit ist S(i,j) nur noch die Summe von ¼ so vielen Werten!
- Man muß dann einfach für die Berechnung von $\sum_{k=i_1}^{i_2} \sum_{l=j_1}^{j_2} T(k,l)$ eine zusätzliche Fallunterscheidung machen

Anwendung von Summed-Area Tables

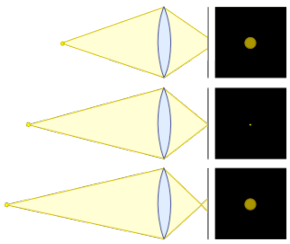
- Generell zum Filtern
- Beispiel: durchscheinende Objekte (*translucency*)
  1. Rendere die Szene ohne die durchscheinenden Objekte
     - Speichere Bild in Textur
  2. Rendere durchscheinende Objekte mit speziellem Fragment-Shader über das bestehende Bild
     - Fragment-Shader gibt einfach Mittelwert innerhalb einer kleinen Region um das aktuelle Pixel aus zuvor gerendertem Bild aus
- Anregung zum Weiterdenken:
  - Wie kann man verschiedene durchscheinende Materialien darstellen? (Milchglas, unebenes Plastik, bunte Folie)
  - Wie könnte man mehrere hintereinander liegende durchscheinende Objekte rendern?

☺ Studienarbeit ☺

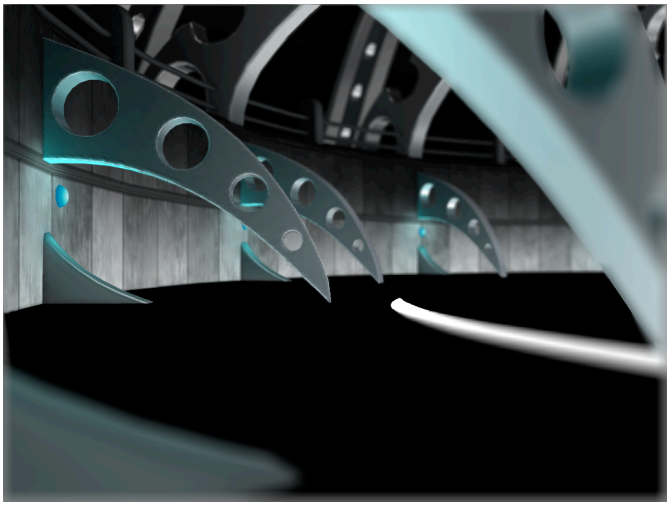
- Resultat:



- Beispiel: Depth-of-Field (Tiefenunschärfe)
  - Rendere die Szene
    - Speichere Color-Buffer und Depth-Buffer in Texturen
  - Berechne Summed-Area Table für Color-Buffer
  - Render screen-filling Quad mit speziellem Fragment-Shader
  - Lese z-Wert aus Depth-Buffer
  - Bestimme daraus den "circle of confusion"
  - Leite daraus Rechteck ab, über das der Color-Buffer gemittelt wird



- Resultat:



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