Computations in Graphics Hardware

basierend auf

Using Graphics Cards for Quantized FEM Computations

Virtual 16 Bit Precise Operations on RGBA8 Textures

von Robert Strzodka
und Martin Rumpf

Seminar „Physically-based Methods for 3D Games and Medical Applications“

Andreas Helbling

Die Autoren

Dipl.-Math. Robert Strzodka

Research Areas

ïHardware sensitive analysis of PDE methods in image processing
ïFast implementations on alternative hardware designs
ïIntegration of data preprocessing and visualization

Prof. Dr. Martin Rumpf

Working group Numerical Analysis and Scientific Computing

‹ bersicht

ï**Motivation**
ï**Idee**
ï**Rechnen**
ï**Matrix-Vektor Produkt**
ï**Anwendung: L^ sen eines Gleichungssystems**
ï**Linear Heat Equation**
ï**Fire...**

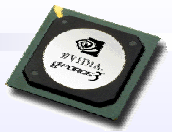
A
B
?

Motivation (1)

Vergleich der Architekturen: GPU vs. CPU

	GeForce 4 Ti4600	Pentium 4
Clock	300 MHz	2-3 GHz
Datenbus	128 Bit	64 Bit
Memory	128 MB	Up to 4 GB
Mem clock	650 MHz DDR	533 MHz QDR
Mem latency	2 or 3 clocks	3 clocks
Bandwidth	10.4 GB/s	4.2 GB/s
Caches	ÑQuad Cacheì, size ?	8kB L1 (data) / 512 kB L2
in Parallel	8 textures / rendering pass	SSE: 4 float ops, MMX: 2 int ops
+	Hohe Bandbreite Schnelles RAM Datenbus ist 128 Bit	CPU Takt sehr hoch Viel RAM Sehr flexibel, Ñkann allesì
−	GPU Takt tief Kann nur spezielle Operationen	Effizienz ~ Cacheverhalten

Motivation (2)



GPU

ïkommt ohne Caches zurecht

ïausgelegt f¸ r grosse Datenmengen

ïRAM ist viel grˆ sser als CPU Cache

ïspezielle Operationen f¸ r grosse Datenmengen

ïh%ufig nicht ausgelastet

CPU

ïnur effizient, wenn das Datenvolumen kleiner ist als der Cache

ïOperationen, nur auf einzelne/wenige Werte anwendbar

ïf¸ r Programme optimiert (Branch-Prediction etc..)

ïmacht alles Andere auch noch

Motivation (3)

Textur-Matrix Analogie

Wie kommt meine GeForce zu Daten ?

ïTextur = $[R_1G_1B_1A_1, R_2G_2B_2A_2, R_3G_3B_3A_3, ...]$

m*n Texel = m*n*4 Farbintensit%ten aus [0, 255]

ïMatrix = $[m_{11}, m_{12}, m_{13}, ...]$

m*n Zahlen aus $[-\infty, \infty]$

Eine Textur ist eine Matrix [von 4-Vektoren], deren Komponenten nur Integer-Werte aus [0, 255] annehmen kˆ nnen

Idee (1)

OpenGL

ïeinfache und direkte Nutzung der Grafikkarte

ïHardware-unabh%ngig

ïGLUT: Plattform-unabh%ngig

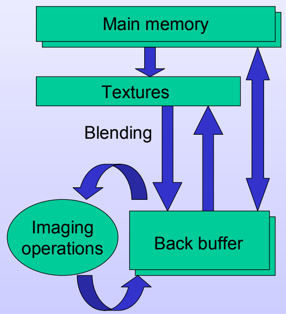
ïauf vielen Plattformen verf¸ gbar

Zentrale Idee: Benutze Blending f¸ r die Implementierung von algebraischen Operationen auf Farbwerten

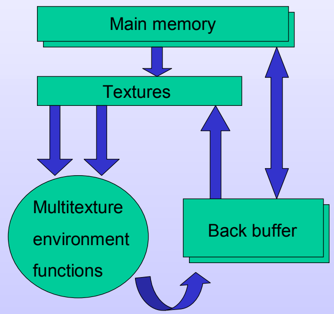
2 Ans%tze: Fragment-basierend vs. Texel-basierend

Idee (2)

Fragment-basierend: Benutze Operationen, die direkt auf Pixeln im Framebuffer operieren

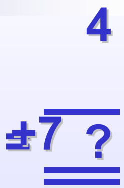


Texel-basierend: Benutze Operationen, die auf Texturen operieren



Rechnen - Einfache Operationen (1)

Fragment-basierend



Operation	Formel	OpenGL
addition	**V**+**W**	BlendFunc
komp.multiplikation	**V•W**	BlendFunc
skalar addieren	**V**+b**I**	BlendFunc
skalarmultiplikation	a**V**	BlendFunc
lineare transf.	a**V**+b**I**	PixelTransfer
funktion	f(**V**)	PixelMap
subtraktion	**V**-**W**	BlendEquation
lineare transf.	a**V**±b**I**	BlendFunc
funktion	f(**V**)	ColorTable
maximum	max(**V**,**W**)	BlendEquation
minimum	min(V,**W**)	BlendEquation
faltung	S*V	ConvolutionFilter
vektor norm	$\|\|V\|\|_{k=1..\infty}$	Histogram
color matrix	C**V**	MatrixMode

$\}$ **OpenGL 1.2** (subtraktion … color matrix)

V, **W** : Texturen; a, b : Skalare; **I**:Identit‰tsmatrix; C: Matrix

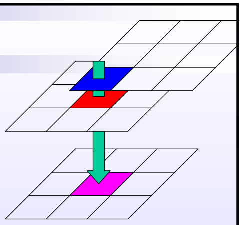
Rechnen - Einfache Operationen (2)

Texel-basierend

Operation	Formel	OpenGL
addition	**V**+**W**	texture_env_add
komp. multiplikation	**V•W**	standard
lineare transf.	a**V**+b**I**	texture_env_add
lineare transf.	a**V**+b**I**	texture_scale_bias
funktion	f(**V**)	texture_color_table
funktion	f(V_0, V_1, V_2)	texture_shader
funktion	f(V_0, V_1, V_2, V_3)	pixel_texture

V, **W** : Texturen; a, b : Skalare; **I**:Identit‰tsmatrix;

Rechnen - Einfache Operationen (3)



BlendFunc

ï‹ berblendung

ïFragmente werden gemischt

The **glBlendFunc** function specifies pixel arithmetic.

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
```

Parameters

sfactor

Specifies how the red, green, blue, and alpha source-blending factors are computed. Nine symbolic constants are accepted: **GL_ZERO, GL_ONE, GL_DST_COLOR,GL_SRC_ALPHA, GL_DST_ALPHA,...**

dfactor

Specifies how the red, green, blue, and alpha destination-blending factors are computed. Eight symbolic constants are accepted: **GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_SRC_ALPHA, GL_DST_ALPHA,...**

Rechnen - Einfache Operationen (4)

BlendFunc Faktoren

```
void glBlendFunc(
        GLenum sfactor,
        GLenum dfactor
);
```

ïbestimmen resultierendes Fragment

ïsind **nicht** beliebig

ïsind 1, 0 oder Farbe resp. Alpha einer der Texturen

ïAlpha-Werte kˆ nnen beliebig in [0..255] sein

ïFarbe/Alpha einer Textur: Faktor **pro Pixel**

Result = sfactor*Source + dfactor*Destination

Source: Fragmente, die neu in den Framebuffer kommen

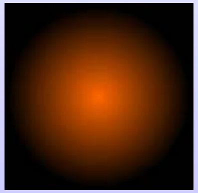
Destination: Fragmente, die bereits im Framebuffer sind

Rechnen - Einfache Operationen (5)

BlendFunc Beispiel: Komponentenweise Multiplikation

```
void glBlendFunc(
        GLenum sfactor,
        GLenum dfactor
);
```

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT);
   glBlendFunc(GL_ONE, GL_ZERO);
   glDrawPixels(256, 256, GL_RGB, GL_UNSIGNED_BYTE, texture);
   glBlendFunc(GL_ZERO, GL_SRC_COLOR);
   glDrawPixels(256, 256, GL_RGB, GL_UNSIGNED_BYTE, spot);
   glFlush();
}
```

 ●  = 

Rechnen - Zahlenformate (1)

Grafikhardware: Intensit‰ten $\in$ [0,255] resp [0,1]

➡ Wie kann man float-Zahlen mit 8Bit Intensit‰ten emulieren ?

r: $[-\rho_0, \rho_1] \rightarrow [0,1]$; $\boxed{r(x) = (x+\rho_0)\frac{1}{\rho_0+\rho_1}}$

r^{-1}: $[0, 1] \rightarrow [-\rho_0, \rho_1]$; $\boxed{r^{-1}(x) = \frac{x}{1}(\rho_0+\rho_1) - \rho_0}$

ïjede Zahl repr‰sentierbar

ïwird nur durch 8 Bit aufgelˆst ! (Dazu sp‰ter mehr...)

ïρ_0 und ρ_1 mˆglichst klein w‰hlen.

ïMeistens will man $\rho_0=\rho_1=\rho$

Rechnen - Zahlenformate (2)

$\boxed{r(x) = (x+\rho_0)\frac{1}{\rho_0+\rho_1}}$

Problem: Wir mˆchten aus den Zahlen Z:[-100,100] den Wert x=50+25=75 auf der Grafikhardware berechnen. Die Intensit‰t des Resultates sollte also

r(75) = (75+100)/(100+100) = 175/200 = 0.875 sein.

r(50) = (50+100)/(100+100) = 150/200 = 0.75

r(25) = (25+100)/(100+100) = 125/200 = 0.625

GPU: 0.75+0.625 = 1.375 → overflow →

Trick: a+b rechnet man so: $\boxed{a+b = 2\left(\frac{1}{2}r(a) + \frac{1}{2}r(b) - \frac{1}{4}\right)}$

→ 2*(1/2*0.75 + 1/2*0.625 - 1/4) = 2*(0.375 + 0.3125 - 0.25)

= 2*(0.6875 - 0.25) = 2*(0.4375) = 0.875

Weitere Formeln f¸r Multiplikation, Lineare Transformation, etc...

Rechnen - 16 Bit genaue Fixpunktzahlen (1)

Idee:

ïVerteile Zahl auf zwei 8Bit Zahlen

ïRechnen wird komplizierter und langsamer

ïImplementation z.B. mit PixelShadern

R	G	B	A
x_1	x_2	x_3	x_4

⬇

R	G	B	A	
x_1		x_2		?
x_1	x_2		x_1	?
x_1	x_2	x_1	x_2	?

Anforderungen an das 16Bit Format:

ïObermenge der existierenden 8Bit / 9Bit Formate

ïHigher 8 Bits = Beste 8Bit Approximation

ïEffiziente Implementation (z.B. carry-over)

Rechnen - 16 Bit genaue Fixpunktzahlen (2)

Definition

R	G	B	A
x_{1hi}	x_2		x_{1lo}

x_1= Ψ(R,A)=ψ(r,a); R,A ∊ [0,255]; r,a ∊ [0,1]

x_2= Ψ(G,B)=ψ(g,b); G,B ∊ [0,255]; g,b ∊ [0,1]

$$\psi(x_{hi}, x_{lo}) = (2x_{hi} - 1) + \frac{1}{128}(x_{lo} - \frac{1}{2})$$

$$\Psi(X_{hi}, X_{lo}) = \frac{1}{255}((2X_{hi} - 255) + \frac{1}{128}(X_{lo} - \frac{1}{128})$$

Dieses Format erf¸llt alle Anforderungen

Rechnen - 16 Bit genaue Fixpunktzahlen (3)

Zusammengefasst:

ïH^here Genauigkeit

ïF¸r spezielle (Teil-)Probleme sinnvoll

ïImplementation kompliziert

ïNur wenige Operationen realisierbar

ïNeue Operationen nur noch halb so schnell, Multiplikation noch langsamer

Besser warten auf GeForce FX

	$\mathbf{tex0[RA, GB] + (s_x, s_y) \cdot tex1[RA, GB]}$
0: RGB	$(tex0[RGB] - \frac{1}{2}) + (s_x, s_y, s_y) \cdot (tex1[RGB] - \frac{1}{2}) \rightarrow tex0[RGB]$
0: A	$tex0[B] + s_y(tex1[B] - \frac{1}{2}) \rightarrow sp0[A]$
1: RGB	$(sp0[A] < \frac{1}{2})?\ -(0, (s_y - 1)/2^9, \frac{1}{2}) : -(0, (s_y+1)/2^9, -\frac{1}{2}) \xrightarrow{-\frac{1}{2}} sp0[RGB]$
1: A	$tex0[A] + s_x(tex1[A] - \frac{1}{2}) \rightarrow sp0[A]$
2: RGB	$tex0[RGB] + (0, -1, -1) \cdot sp0[RGB] \rightarrow tex0[RGB]$
2: A	$(tex0[A] - \frac{1}{2}) + s_x(tex1[A] - \frac{1}{2}) \rightarrow tex0[A]$
3: RGB	$(sp0[A] < \frac{1}{2})?\ -((s_x - 1)/2^9, 0, \frac{1}{2}) : -((s_x+1)/2^9, 0, -\frac{1}{2}) \xrightarrow{-\frac{1}{2}} sp0[RGB]$
4: RGB	$tex0[RGB] + (-1, 0, 0) \cdot sp0[RGB] \rightarrow tex0[RGB]$
4: A	$tex0[A] + (-1) \cdot sp0[B] \rightarrow tex0[A]$

Rechnen - Optimierungen (1)

Blending braucht immer mindestens 2 rendering passes

OpenGL Erweiterung `EXT_texture_env_combine` und Texturfunktion `ADD_SIGNED_EXT`

$a + b = 2\left(\frac{1}{2}r(a) + \frac{1}{2}r(b) - \frac{1}{4}\right)$ in einem rendering pass

Allgemeiner: Erweiterung `NV_register_combiners`

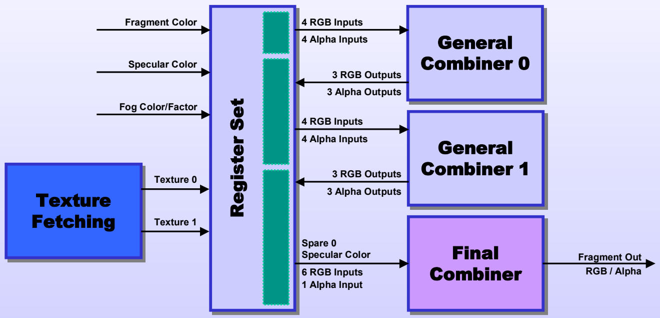
ïAddition der Intensit‰ten implementierbar

ïZwischenresultate der Combiners k^nnen in [-1,1] sein

ïAddition in einem rendering pass

Rechnen - Optimierungen (2)

Register Combiners



Matrix-Vektor Produkt (1)

ÑZeile mal Spalteì - und zwar komponentenweise

$$A\vec{x} = \vec{y}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots \\ a_{21} & a_{22} & a_{23} & \dots \\ a_{31} & a_{32} & a_{33} & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \end{bmatrix} \Rightarrow y_i = \sum_j \begin{bmatrix} a_{11} * x_1 \\ a_{12} * x_2 \\ a_{13} * x_3 \\ \dots \end{bmatrix}_j = \sum_j (\vec{a}_i \bullet \vec{x})_j = \|\vec{a}_i \bullet \vec{x}\|_1$$

ïKomponentenweise Multiplikation von Vektoren

ïAufsummieren der Komponenten

Mˆ gliche Implementation:

ïA als Menge von Texturen und **x** als einzelne Textur

ïKomponentenweise Multiplikation mit BlendFunc

ï1-Norm u.a. mit Histogramm

Anwendung: Lˆ sen eines Gleichungssystems

A**x**=**b**

Lˆ sen mit iterativem Algorithmus: $\mathbf{x}_{i+1}$=F(x_i); $\mathbf{x}_0$=**b**;

ïJacobi-Iteration

ïKonjugierte Gradienten

Methoden benutzen nur Operationen, die wir schon gesehen haben

Allgemein: Fast alles ist machbar, nur Frage von Aufwand, Effizienz und Genauigkeit

Linear Heat Equation (1)

Zeitabh‰ngige Temperaturverteilung u(**x**,t)

Anfangsverteilung u_0 = u(t = 0)

Heizungen resp. K¸ hlkˆ rper f(**x**)

$$\frac{\partial}{\partial t}u - \Delta u = f$$

Funktion f als diskretisierte Funktion in einer Textur F gespeichert:

f(**x**)=f(x_1,x_2)=F[x_1,x_2]   (look-up table)

Linear Heat Equation (2)

Lˆ sung durch iterative Methode in jedem Zeitschritt

Pseudocode:

```
HeatEquation(u0, f, λ)
Load Images representing u0, f and parameters into Textures
// from now on, everything is done in graphics hardware
for each timestep k
{
        calculate R^k =U^k+λF
        Init iterative solver with S^0=R^k
        for each iteration i
        {
                calculate S^i+1
        }
        store U^k+1=S^i+1
}
```

Fire...

Einfach(st)er Algorithmus f¸ r 2D Feuer:

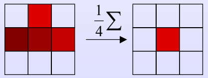
1 Addiere an N zuf‰lligen (x,y) in untersten zwei Zeilen einen Wert

2 Img[x,y]=(Img[x,y]+Img[x-1,y]+Img[x+1,y]+Img[x,y+1])/4

3 Img[x,y]>0 → Img[x,y]--;

4 Zeichne Img (Feuer-Palette)

5 Goto 1



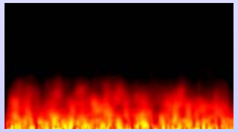
Machbar in Hardware

ïZufallstextur addieren mit BlendFunc

ïSumme mit BlendFunc

ïImg[x,y]-- mit zu kleinen Alpha-Werten

ïPalette mit PixelMap



Zusammenfassung

ïNeue Perspektive

ïGrafikhardware kann grosse Datenmengen schnell und parallel verarbeiten

ïWichtige Grundoperationen realisierbar

ïGrafikkarte als Coprozessor

ïF¸ r gewisse Anwendungen geeignet

ïAuch komplexere Probleme lˆ sbar (PDEís)

ïRelativ kompliziert und beschr‰nkt f¸ r wissenschaftliche Berechnungen

ïHardware wird immer Ñprogrammierbarerì

Ausblick, Diskussion

Nachteile

ïOperationen m¸ ssen Vorteile der Grafikhardware nutzen → Approximierung von nicht-linearen Funktionen durch lineare

ïIntervall grˆ sser als [-1,1] → PixelTransfer Funktion, ist aber langsam und eingeschr‰nkt

ïglHistogram f¸ r Zwischenresultate zu langsam

ïGenauigkeit limitiert Anwendungsmˆ glichkeiten

Ausblick

ïPixelShadern und RegisterCombiners auf GeForce 3/4

ïGeForce FX: 128Bit Farben, d.h. 32 Bit pro Kanal, erweiterte PixelShader