Nonlinear Diffusion in Graphics Hardware

Based on the homonymous paper by
Martin Rumpf and Robert Strzodka

Pascal Gwosdek

13.06.07

Diffusion

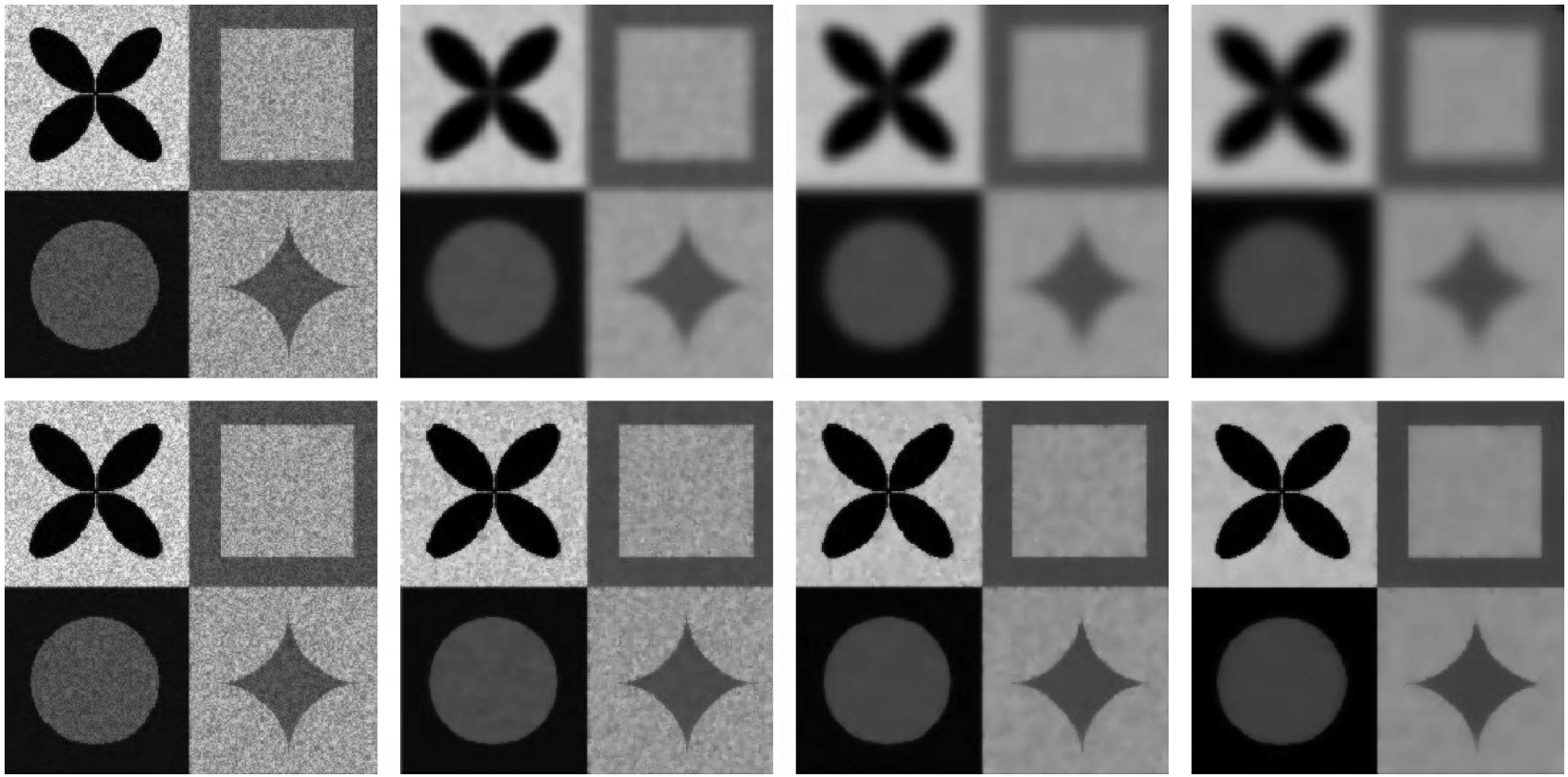


Figure: Linear (top) and nonlinear (bottom) diffusion. **Authors:** M. Rumpf and R. Strzodka.

Graphics Hardware

- Today: GPUs superior to CPUs in computing power
- But: GPUs specialized
  - Restricted range of values
  - Less accuracy
  - Restricted set of instructions
  - Parallelism dominates sequential execution by design

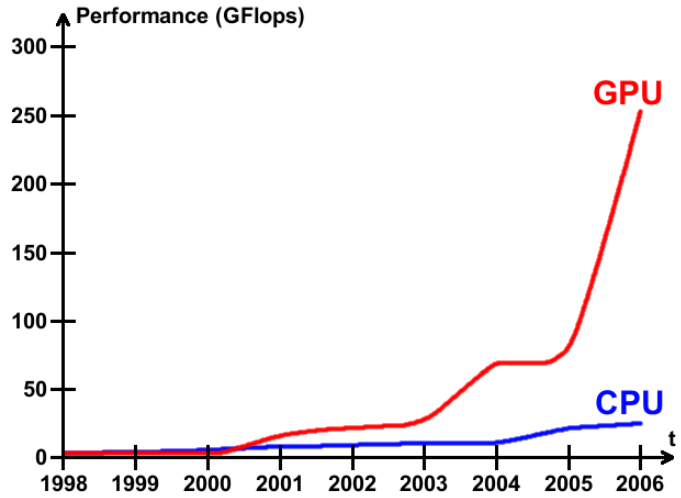


Figure: Performance of ATI/AMD GPUs and CPUs. **Authors:** wired.com, Pascal Gwosdek

OpenGL Rendering pipeline

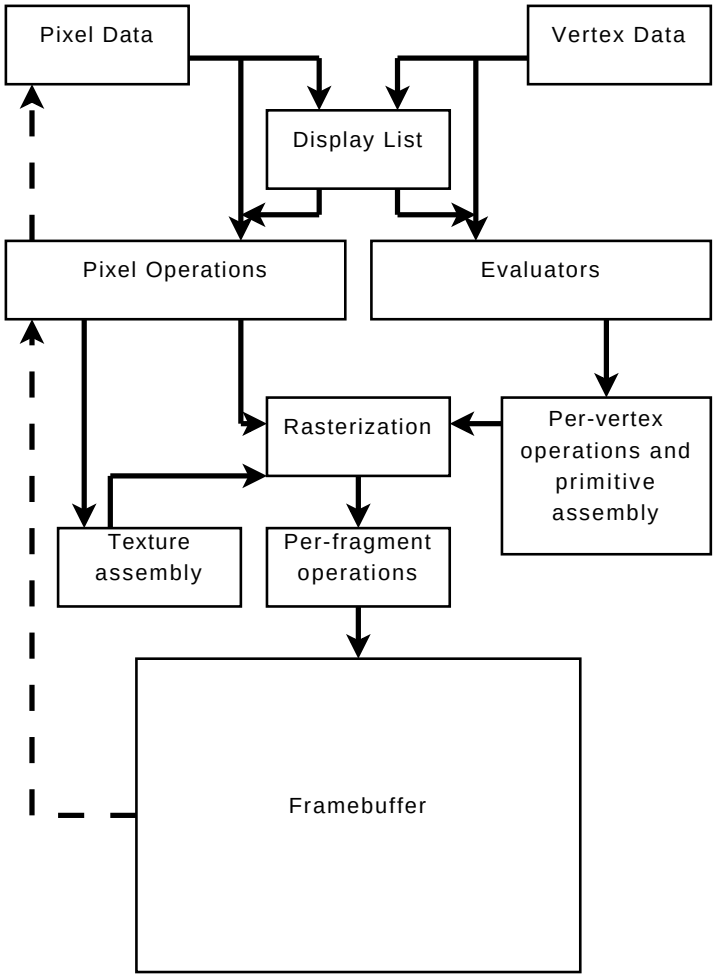


Figure: The OpenGL rendering pipeline. **Authors:** OpenGL Architecture Review Board, Pascal Gwosdek

OpenGL Rendering pipeline (simplified)

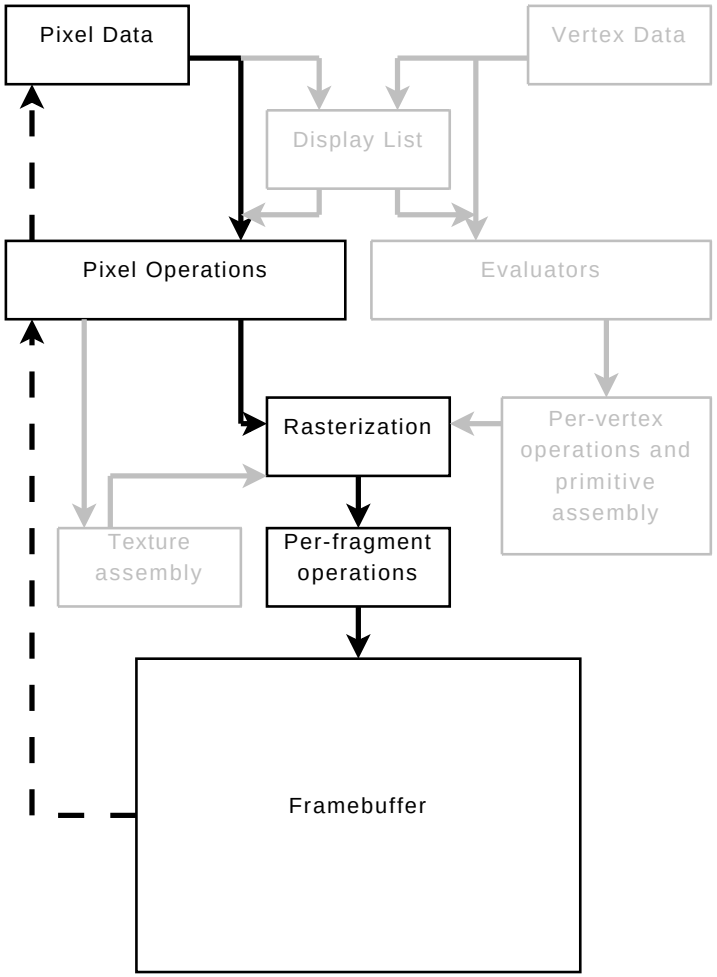


Figure: The OpenGL rendering pipeline. **Authors:** OpenGL Architecture Review Board, Pascal Gwosdek

Operations

Figure: Linear combination. **Authors:** MIA Group, Informationsdienst Wissenschaft e.V., Pascal Gwosdek

Linear combination by `glBlendFunc`:

$$\text{result} = 0.3 \cdot \vec{\text{Weickert}} + 0.7 \cdot \vec{\text{Bruhn}}$$

Operations

Figure: Multiplication. **Authors:** MIA Group, Informationsdienst Wissenschaft e.V., Pascal Gwosdek

Componentwise multiplication by `glBlendFunc`:

$$\text{result}_{ij} = \vec{\text{Weickert}}_{ij} \bullet \vec{\text{Bruhn}}_{ij}$$

Operations

Figure: Function application. **Authors:** MIA Group, Pascal Gwosdek

Function application by `glPixelMap`:

$$\text{result}_{ij} = F(\vec{\text{Weickert}}_{ij})$$

$$\text{result} = F \circ \vec{\text{Weickert}}$$

Operations

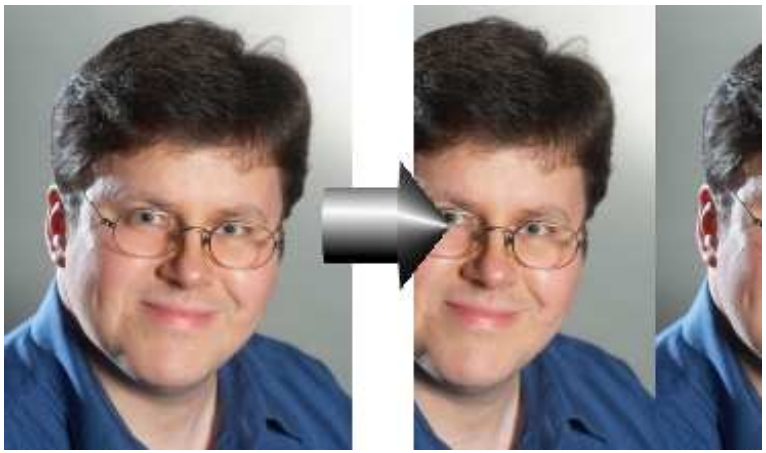


Figure: Index shift. **Authors:** MIA Group, Pascal Gwosdek

Index shift by change of drawing positions:

$$\text{result} = T_\gamma \circ \vec{\text{Weickert}}$$

Operations

Figure: Histogram for vector norm. **Authors:** MIA Group, Pascal Gwosdek

With histograms provided by `glGetHistogram`, vector norms are easier to compute:

$$||\vec{\text{Weickert}}||_k = \left(\sum_{x=0}^{n} x^k \cdot H(x) \right)^{\frac{1}{k}}$$

Operations

Inner products can be written as norm of componentwise vector products:

$$\langle \vec{X}, \vec{Y} \rangle = ||\vec{X} \bullet \vec{Y}||_1$$

Operations

Vector matrix products can be computed by

$$A\vec{X} = \sum_{\gamma=0}^{n-1} T_\gamma \circ (\vec{A^\gamma} \bullet \vec{X}).$$

with $\vec{A^\gamma}$ as the composed diagonal vector of $A = (a_{ij})_{ij} \in \mathbb{R}^n \times \mathbb{R}^n$ s.t. $\gamma = j - i \mod n$.

Operations

Example: Vector matrix product

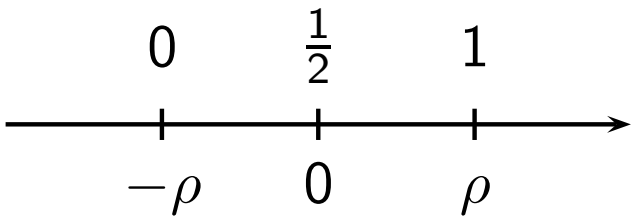
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 \\ 4 \cdot 1 + 5 \cdot 2 + 6 \cdot 3 \\ 7 \cdot 1 + 8 \cdot 2 + 9 \cdot 3 \end{pmatrix}$$

$$= \begin{pmatrix} 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 \\ 5 \cdot 2 + 6 \cdot 3 + 4 \cdot 1 \\ 9 \cdot 3 + 7 \cdot 1 + 8 \cdot 2 \end{pmatrix}$$

$$= T_0 \circ \begin{pmatrix} 1 \cdot 1 \\ 5 \cdot 2 \\ 9 \cdot 3 \end{pmatrix} + T_1 \circ \begin{pmatrix} 7 \cdot 1 \\ 2 \cdot 2 \\ 6 \cdot 3 \end{pmatrix} + T2 \circ \begin{pmatrix} 4 \cdot 1 \\ 8 \cdot 2 \\ 3 \cdot 3 \end{pmatrix}$$

Numbers

Graphics hardware is restricted to a fixed point number range from 0 to 1. This turns out to be a technical issue only.



$\longrightarrow$ perform linear scaling

Perona-Malik

The authors use the modification of the Perona-Malik model proposed by Catté, Lions, Morel, and Coll:

$$\begin{aligned} \partial_t u - \operatorname{div}\left(g\left(\nabla u_\varepsilon\right)\nabla u\right) &= 0, && \text{in } \mathbb{R}^+ \times \Omega, \\ u\left(0,\cdot\right) &= u_0, && \text{on } \Omega, \\ \frac{\partial}{\partial \nu} u &= 0, && \text{on } \mathbb{R}^+ \times \partial\Omega. \end{aligned}$$

Discretization

The model is discretized in a semi-implicit manner using forward difference and applying the finite element method, which finally yields

$$\left(I + \frac{\tau}{h^2} \left(\left\langle g\left(\nabla \overrightarrow{U_\varepsilon^k} \right) \nabla \widehat{\Phi_\alpha}, \nabla \widehat{\Phi_\beta} \right\rangle \right)_{\alpha\beta} \right) \overrightarrow{U^{k+1}} = \overrightarrow{U^k},$$

This formula constitutes a linear system of equations of the form

$$A^k \left(\vec{U^k} \right) \vec{U^{k+1}} = \vec{R} \left(\vec{U^k} \right).$$

Solving the equation system

Two iterative solvers for the linear equation system are compared:

- The **Jacobi Iteration**

$$F\left(\vec{X}^i\right) = D^{-1}\left(\vec{R} - (A - D)\,\vec{X}^i\right),$$

  with D as the diagonal of A and
- the **Conjugate Gradient (CG)** method, which is based on a minimization problem for

$$\Phi(\vec{X}) = \frac{1}{2}\vec{X}^T A\vec{X} - \vec{X}^T R.$$

Algorithm

nonlinear diffusion
{
load image, parameters
init hardware
encode image $\vec{U}^0$

forall (timesteps k)
{
store $\vec{R}^k \leftarrow \vec{U}^k$
calc diffusivity $\rightsquigarrow A$
init solver $\vec{X}^0 \leftarrow \vec{U}^k$
forall (iterations l)
calc $\vec{X}^{l+1} \leftarrow F\left(\vec{X}^l\right)$
store $\vec{U}^{k+1} \leftarrow \vec{X}^{l+1}$
decode $\vec{U}^{k+1}$ and display
}
}

Results

Figure: Comparison of different solvers. **Top:** Adaptive software preconditioned CG [sic!]. **Middle:** Jacobi solver on graphics hardware. **Bottom:** CG solver on graphics hardware. **Authors:** M. Rumpf and R. Strzodka.

Results

Figure: Comparison of different solvers (Zoom). **Left:** Adaptive software preconditioned CG [sic!]. **Middle:** Jacobi solver on graphics hardware. **Right:** CG solver on graphics hardware. **Authors:** M. Rumpf and R. Strzodka.

Results

Method		t / 100 iterations
CG solver	in software	"fast"
Jacobi solver	on NVidia GeForce2 Ultra	$\approx$ 1.7 sec
Jacobi solver	on SGI Onyx2	17 sec
CG solver	on SGI Onyx2	42 sec

Table: Comparison of different solvers, for 256 $\times$ 256 px images. **Note:** The authors do not mention the time of the reference implementation, but speak of "surprisingly weak performance" of the "slower" hardware implementations on the SGI Onyx. **Authors:** M. Rumpf and R. Strzodka.

Discussion

Possible reasons for the weak performance:

- read back of framebuffer about 60 times slower than writing
- `glGetHistogram` slow

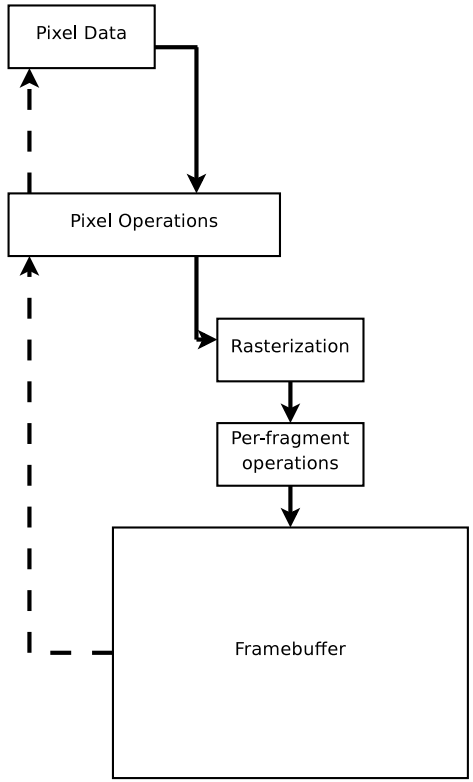


Figure: Excerpt of the OpenGL rendering pipeline.
Authors: OpenGL Architecture Review Board, Pascal Gwosdek

Summary

- Linear algebra in graphics hardware
- Nonlinear diffusion
  - Overview
  - Implementation
- Results
- Problems

References


OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis.
OpenGL Programming Guide.
http://www.opengl.org/documentation/red_book/.


E. Bruce Pitman.
Conjugate Gradient Method.
http://www.ccr.buffalo.edu/class-notes/hpc2-00/odes/node4.html, 2000.


Martin Rumpf and Robert Strzodka.
Nonlinear Diffusion in Graphics Hardware.
In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 75–84. Universität Bonn, Springer, 2001.


Robert Strzodka.
Hardware Efficient PDE Solvers in Quantized Image Processing.
PhD thesis, Fachbereich Mathematik der Universität Duisburg-Essen, 2004.


Wikipedia.
Finite Element Method.
http://en.wikipedia.org/wiki/Finite_element_method, 2007.

Discussion

Thank you!

Please feel free to ask questions.

Solving the equation system - The CG method

The CG solver is given by

$$F\left(\vec{X^i}\right) = \vec{X^i} + \frac{\left\langle \vec{r^i}, \vec{r^i} \right\rangle}{\left\langle A\vec{p^i}, \vec{p^i} \right\rangle} \cdot \vec{p^i},$$

with the search directions

$$\vec{p^i} = \vec{r^i} + \frac{\left\langle \vec{r^i}, \vec{r^i} \right\rangle}{\left\langle \vec{r^{i-1}}, \vec{r^{i-1}} \right\rangle} \cdot \vec{p^{i-1}}$$

and the residuals

$$\vec{r^i} = \vec{R} - A\vec{X^i}.$$