In *Proceedings VIIP'01*, pages 193–202, 2001.

Using Graphics Cards for Quantized FEM Computations

MARTIN RUMPF
Deparment of Applied Mathematics
University of Duisburg
47048 Duisburg, Germany
email: rumpf@math.uni-duisburg.de

ROBERT STRZODKA
Deparment of Applied Mathematics
University of Duisburg
47048 Duisburg, Germany
email: strzodka@math.uni-duisburg.de

ABSTRACT
Graphics cards exercise increasingly more computing power and are highly optimized for high data transfer volumes. In contrast typical workstations perform badly when data exceeds their processor caches. Performance of scientific computations very often is wrecked by this deficiency. Here we present a novel approach by shifting the computational load from the CPU to the graphics card. We represent data in images and operations on vectors in graphics operations on images. Broad access to graphics memory and parallel processing of image operands thus turns the graphics card into an ultrafast vector coprocessor. The presented strategy opens up a wide area of numerical applications for hardware acceleration. The implementations of Finite Element solvers for the linear heat equation and the anisotropic diffusion method in image processing underline its practicability.
We explain the vector processor usage of graphics cards in detail. An extensive correspondence of vector and graphics operations is given and the decomposition of complex operations into hardware supported is explicated. We also sketch the realization of arbitrary number formats in graphics hardware and the consequences of the restricted precision. Finally, we propose slight modifications and extensions which would further improve computational benefits and extend the range of applicability of the proposed approach. Computing in image processing at 5ms for an Jacobi iteration on 128^2 images is exemplarily depicted as an ideal field, where Finite Element methods can be greatly accelerated and ultimate number precision is not required.

KEY WORDS
graphics hardware computing, hardware accelerated numerical methods, anisotropic diffusion

1. Introduction

In the last two decades PC graphics hardware has developed dramatically boosting its performance, functionality and programmability. The former line drawer has become a graphics processor unit (GPU), which outrivals the CPU in increasingly many computations. This enormous success could only be accomplished so quickly, because graphics hardware development closely followed the needs of graphics programmers, whereas the general purpose microprocessors could not be oriented solely towards graphics requirements. In the continuation of this development there is now a very good opportunity to upvalue the GPU to a revolutionary fast vector coprocessor.

Motivation

In the last years graphics hardware design has been particularly sensitive to memory bandwidth problems due to a rapidly increasing data transfer volume. As a result modern GPUs can access and transfer large data blocks tremendously faster than the CPU.
In micro-processors the same problem has namely been tackled by introducing a hierarchy of fast memory caches, which is very good for accelerating repeated random memory accesses, but fails for large data blocks which exceed the cache sizes. Unfortunately most scientific applications have to handle large amounts of data and thus they suffer from both the limited main memory bandwidth and the obsolete repeated command transfer, when the same operation has to be performed on each component of the data block. Therefore many typical scientific computing applications perform at about 1% of the peak processor performance. This disastrous situation is still little acknowledged. Though there are successful concepts how to significantly optimize storage and access for caching [4, 16, 2], overall performance remains far away from peak values. Unfortunately this will not change in near future, because cache sizes are miles away from reaching the size of graphics memory, and even then they would still lack the vector operations on entire data blocks. In conclusion we see that when competing with typical micro-processor systems graphics boards are much better suited for scientific applications dealing with regular large data blocks.
Many implementations of this philosophy, though they did not always explicitly subscribe to it, have already bore very fruitful results. In particular volume rendering, including lighting and shading, has greatly benefited from the exploitation of fast texturing and blending operations [18, 5, 10]. But also further going techniques of image

analysis and filtering have found support in graphics hardware functionality [11, 6, 7] and even a rather complex application like vector field visualization has been implemented [3, 8].

Goals

Going beyond rendering calculations and image transformations we want to show that the functionality of modern graphics cards has reached a state, where the graphics processor unit may be regarded as a programmable fixed-point vector coprocessor. Anything from basic algebraic operations to arbitrary functions of several variables can be mapped onto graphics hardware functionality. Observing the precision restrictions even typical discrete numerical schemes for partial differential equations can be implemented completely in graphics operations.

We will show how this coprocessor usage can be accomplished and will demonstrate its capacity and flexibility by implementing Finite-Element schemes for the linear heat equation and the anisotropic diffusion model [17], used for the edge sensitive denoising of images. But these applications really only scratch the surface of the looming possibilities.

The main reason for bringing numerical computations to graphics hardware is the formerly explained unrivaled dominance of the GPU over the CPU in data transfer dominated applications. Also, processor design orientates towards general software optimizations comprising branch predicted execution, fast local computations and cached random memory access. It is hard to exploit these in solvers for partial differential equations modeling various processes on 2D or 3D domains. They would rather need vector operations on entire data blocks and fast broad access to large amounts of data. These are issues also put forward -though maybe in different terms- by the graphics community. Moreover, recent advances in graphics hardware functionality clearly convey the tendency to arbitrary algebraic operations and pipeline customization, which are both very beneficial for numerical scheme implementations. Certainly there are some obstacles like the restricted number formats and precision or some unoptimized parts of the graphics pipeline, but the overall hardware design and development amazingly fits the numerical purpose.

Therefore, our primary goal here is to support a dialog across the disciplines of graphics hardware development and scientific computing, which, with little effort on both sides, could achieve astounding results. The applications presented, demonstrate the huge potential in performance lurking in these graphics hardware based implementations. But many numerical algorithms still disregard hardware issues and little humps in the graphics hardware still obstruct the passage to general fast numerical computations. Hence even minor considerations of graphics hardware issues with respect to numerics on the one side, and development of slightly more hardware sensitive algorithms on the other, could result in revolutionary speedups for many applications. We hope that the forthcoming benefits will raise interest in this interdisciplinary field on both sides.

Since our approach tries to bring together both concepts of numerical analysis and graphics programming, familiarity with one of these areas surely gives a different perspective on the subject. We think that an alternating change of the perspective would leave everyone unsatisfied with the presentation, so here we have chosen to focus on the graphics perspective. To span the bridge from numerics to graphics programming we must, however, utilize mathematical language dealing with partial differential equations and its numerical treatment.

The present paper complements and greatly expands our recent presentation at VisSym'01 ([14]). In [14] we had to restrict ourselves to a preliminary study of possibilities on older graphics hardware, and have dealt only with the basic Perona Malik diffusion model. Here we present a thorough exposition of graphics feature exploitation for numerical computations, and have expanded the image processing to the full anisotropic diffusion model. Finally, the new hardware supported multitexture based implementation here redeems the formerly drawn up prospect of effective speedup results on nowadays PCs.

2. Computational Setting

Here we explain roughly in which ways we intend to use the graphics card for computations. The implementations of our applications are based on the OpenGL API [12], and to provide accuracy we will refer to OpenGL commands in the text, but naturally the framework presented is independent of the API.

We assume a modern graphics card with designated texture memory of high memory bandwidth and an equally fast GPU. In the GPU effectively only the rasterization engine is strained, since no other primitives than textured quads of fixed size are drawn. The texture memory should be able to store the algorithm dependent number of auxiliary image operands of the requested size, e.g. $128^2 * 3 * 16 = 0.75$MB of texture memory for 128^2 RGB images in case of the anisotropic diffusion, discussed in Section 7..

The basic idea of the computing is to use the blending capacities and the imaging subset or the texture environment functions and its extensions to perform algebraic operations on images. We will call the first approach the *fragment based implementation* and the second the *texel based implementation*. In both cases the texture memory is used to contain an array of images, which hold the initial, intermediate and final data of the calculations.

In the fragment based implementation the back buffer is used to combine intermediate results through blending and to apply extended image operations via *glCopyPixels*. In the texel based implementation the multitexture extension is needed to enable the combination of several textures and the result is stored in another texture. Since currently one cannot render directly to a texture, the back buffer is used as a temporary target, from which *glCopyTexSubImage* trans-

fers the result to the target texture. Figure 1 gives an visual overview of the settings.

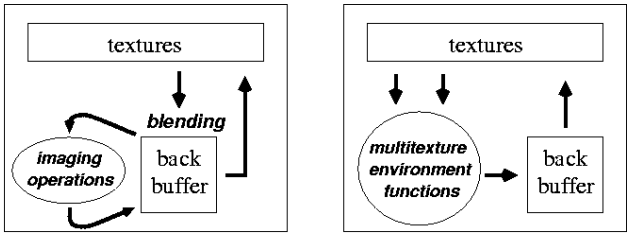

Figure 1. On the left the setting of the fragment based implementation using blending functions and the imaging subset. On the right the texel based implementation using multitextures with texture environment functions and its extensions.

Let us still sketch the overall data flow during program execution. In the beginning initial data is loaded to main memory and then transfered to texture memory. Once the computation has started all operations take place on the graphics card and there is no image transfer to or from the main memory whatsoever. The program only sends graphics commands and parameters to the graphics card, of which the largest are one dimensional textures or lookup tables, necessary to code nonlinear functions. The only data sent back to the main memory are the intensities provided by *glGetMinMax* or *glGetHistogram* if the application chooses to exercise adaptive control.

3. Vector Operations

This section describes the key issues of representing numerical data in images and numerical vector operations by graphics operations on images.

3.1 Vector Representation

In what follows we will discuss numerical schemes for solving partial differential equations. These numerical schemes, however, operate on vectors which describe functions on a given grid, whereas our graphics card operates on images. We therefore have to explain how our images represent these vectors and how our grids look like.

In general the components of the vectors, which describe discrete functions over a grid, are not necessarily the values of the corresponding analytical functions at the grid nodes. But here we deal with the simplest case of an equidistant 2D grid and bilinear Finite Element discretizations, where this is the case.

So our domain of interest Ω is covered by an equidistant $n \times n$ grid. An analytical function u over this domain is approximated by a discrete bilinear function U, which is described by the n^2 values at the grid-nodes. These values are stored in a nodal vector $\bar{U}$ on which the numerical schemes operate. Throughout the paper we will continue to denote the analytical functions by small roman letters (u), the discrete by capital roman letters (U) and the corresponding vectors with an additional bar ($\bar{U}$). Matrices operating on these vectors will be marked by two bars ($\bar{\bar{A}}$).

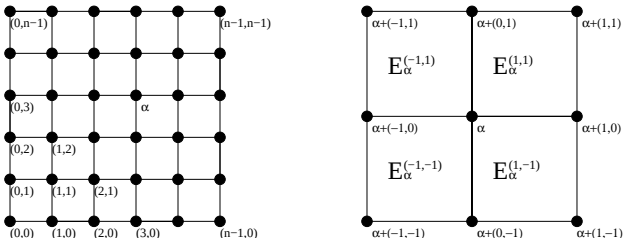

Figure 2. On the left an equidistant $n \times n$ grid enumerated by a tuple, on the right the neighboring elements of a node and the local offsets to neighboring nodes.

Our goal now is to represent the n^2 values of the vector $\bar{U}$ in an image. For this purpose we choose the image to have the same width and height in pixels as our grid in nodes and we enumerate the grid-nodes with a 2-dimensional index $\alpha = (\alpha_x, \alpha_y) \in (0, \ldots, n-1) \times (0, \ldots, n-1)$ according to their x and y coordinates (Fig. 2). This way our image represents the vector $\bar{U}$, such that the vector's component $\bar{U}_\alpha$ is simply the intensity of the pixel at the coordinates (α_x, α_y) in the image. If we deal with vector-valued functions then the vector component $\bar{U}_\alpha$ is itself a small vector, which we represent as the RGBA color vector of the corresponding pixel in a color image. For vector-valued functions with more than 4 components we have to use several images.

To some readers this representation may at first seem like an obsolete tautology. In fact, one could reinterpret all the needed numerical results talking about images and pixel coordinates instead of vectors and their components, but in the long run this mingling of implementational aspects and numerical notation obscures more than it reveals.

In image processing applications, like the anisotropic diffusion presented in Section 7., the input and output data is often itself an image. But in general, the input and output images in such models, are first interpreted as a function over a domain and then again discretized, leading to vectors and their image representations different from the original image.

3.2 Basic Operations

After establishing a firm connection between vectors and images, we may now turn towards the issue of hardware supported vector operations.

First let us consider the componentwise algebraic operations, i.e. operations which act in the same manner on every component of the vector independently of the other vector components. These comprise addition, multiplication,

linear transformation and arbitrary componentwise functions. We have already mentioned the two basically different implementational approaches to realize these operations on images. The fragment based implementation uses the blending capacities and the imaging subset, whereas the texel based implementation assumes a multitexture extension and uses the texture environment functions and its extensions.

Table 1 lists the correspondence of the vector operations to OpenGL functionality for the first implementational approach. The OpenGL function given in the right column of the table naturally only indicates the relevant part of the API, usually additional calls to other functions for execution and further parameter specification are necessary for the implementation of the desired operation. The upper half of the table deals with unextended functionality and we see that even in OpenGL 1.0 we may implement all the basic operations, although hardware support is very rare for the pixel transfer operations. The lower half lists additional and alternative functions while using the imaging subset of OpenGL 1.2 or equivalent extensions. These also include not componentwise functions: the convolution, the vector norms and the color matrix multiplication which operates independently on the vector components but intermingles the components' color-components. We shall discuss them in more detail later in Sections 3.4 and 4.

In Table 2 we give the correspondence of the vector operations to OpenGL functionality for the texel based implementational approach. In the right column this time we list the names of the extensions providing the necessary functions. The upper half of the table is a subset of the functions from Table 1, whereas the lower half introduces functions of several variables through multidimensional dependent texture lookups, which we will consider in more detail in Section 3.4.

Table 1. Correspondence of the blending and imaging subset operations of OpenGL to vector operations, where vectors are represented by images.

operation	**formula**	**OpenGL**
OpenGL 1.0		
addition	$\bar{V} + \bar{W}$	BlendFunc
multiplication	$\bar{V} \bullet \bar{W}$	BlendFunc
scalar add.	$\bar{V} + b\bar{1}$	BlendFunc
scalar mult.	$a\bar{V}$	BlendFunc
lin. trans.	$a\bar{V} + b\bar{1}$	PixelTransfer
function	$f(\bar{V})$	PixelMap
imaging subset of OpenGL 1.2		
subtraction	$\bar{V} - \bar{W}$	BlendEquation
lin. trans.	$a\bar{V} \pm b\bar{1}$	BlendFunc
function	$f(\bar{V})$	ColorTable
maximum	$\max(\bar{V}, \bar{W})$	BlendEquation
minimum	$\min(\bar{V}, \bar{W})$	BlendEquation
convolution	$S * \bar{V}$	ConvolutionFilter
vector norms	$\|\bar{V}\|_{k=1,\ldots,\infty}$	Histogram
color matrix	$C \cdot \bar{V}$	MatrixMode

3.3 Number Formats

The number formats processed in graphics cards have been designed to describe intensities of colors, so in general they only represent the range $[0, 1]$. Although absolute values in scientific computations are not relevant for the processing, after rescaling them we will usually still have to deal with a bigger value range, say $[-\rho_0, \rho_1]$. Therefore we must explain how to emulate number operations on this interval using only intensities from $[0, 1]$.

For finite ρ_0, ρ_1 there is a canonical linear correspondence $r : [-\rho_0, \rho_1] \to [0, 1]$, $r(x) := \frac{1}{\rho_0+\rho_1}(x + \rho_0)$, which we can use to represent numbers from $[-\rho_0, \rho_1]$ by intensities from $[0, 1]$. We have to keep in mind that the intensities from $[0, 1]$ are typically resolved by only 8 bits in graphics hardware, so that the numbers from $[-\rho_0, \rho_1]$ will also be resolved by only 8 bits through our correspondence. It is therefore very important to choose ρ_0 and ρ_1 as small as possible, potentially through the expansion or collapse of formulas and their rearrangement in the algorithm.

To clearly express which entities we mean, we will continue to refer to the elements of $[-\rho_0, \rho_1]$ as the *numbers* or *values* in a texture or framebuffer, and the elements of $[0, 1]$ as the *intensities* in a texture or framebuffer. Similarly we will call the transformation from numbers to intensities the *encoding* and the inverse transformation from intensities to numbers the *decoding*. So by these transformations we obtain encoded numbers and decoded intensities.

Now we know how to represent numbers in a bigger interval but since the GPU will only perform operations on the intensities, we have to define formulas which perform the desired operation on numbers by combining the intensities of the operands into an intensity which represents the corresponding number result. In Table 3 we list exactly these formulas for the symmetric interval $[-\rho, \rho]$. The left column shows the operation to be performed, and the right column shows which operation must be performed on the encoded operands to obtain the equivalent encoded result.

The formulas must be build up and evaluated in a way which guarantees that any intermediate results in intensities do not transcend the range $[0, 1]$, although within certain stages of the graphics pipeline this may happen. For example: $\frac{\rho}{2}$ is represented by $\frac{3}{4}$; the result of the operation on numbers $\frac{\rho}{2} + \frac{\rho}{2} = \rho$ is thus obtained by first comput-

Table 2. Correspondence of the multi-texture environment functions and its extensions in OpenGL to vector operations, where vectors are represented by images.

operation	**formula**	**OpenGL**
addition	$\bar{V} + \bar{W}$	texture_env_add
multiplication	$\bar{V} \bullet \bar{W}$	standard
lin. trans.	$a\bar{V} + b\bar{1}$	texture_env_add
lin. trans.	$a\bar{V} + b\bar{1}$	texture_scale_bias
function	$f(\bar{V})$	texture_color_table
function	$f(\bar{V}_0, \bar{V}_1, \bar{V}_2)$	texture_shader(2)
function	$f(\bar{V}_0, \bar{V}_1, \bar{V}_2, \bar{V}_3)$	pixel_texture

ing $h := \frac{1}{2} \cdot \frac{3}{4} + \frac{1}{2} \cdot \frac{3}{4} = \frac{3}{4}$ which is still in $[0, 1]$ and then $2h - \frac{1}{2} = 1$ which in fact represents the correct number result ρ. For some operations we possibly cannot guarantee the final result in intensities to fit into $[0, 1]$, as for example the product of two numbers from $[-\rho, \rho], \rho > 1$ may always transcend $[-\rho, \rho]$. Therefore it is important to analyze the underlying algorithm in advance and choose ρ appropriately.

We have confined ourselves to a symmetric interval because this is the typical number range of many numerical schemes, and more importantly we may always multiply with -1 without exceeding our interval. Moreover, the encoded operations on intensities become simpler and thus faster with the symmetric encoding.

Finally, we want to emphasize that no other operations than those already discussed in the last Section 3.2 are needed to evaluate the above formulas.

3.4 Optimized Operations

Although we do not need any additional operations to evaluate the formulas from Table 3, which insure the correct functionality for the encoded numbers, we may want to make use of special graphics features to reduce the number of passes necessary for their evaluation. We have several options available.

The form in which the formula for the multiplication has been written in Table 3, for example, already suggests, that the use of appropriate blending source and destination factors, will evaluate $(r(a)(1 - r(b)) + r(b)(1 - r(a)))$ at once and so reduce the number of rendering passes to two. As demonstrated in the last Section 3.3 in the example, the formula for the addition must have a rather awkward form to ensure that the intermediate result after the first blending remains within the range $[0, 1]$. However, the availability of the *EXT_texture_env_combine* extensions with the *ADD_SIGNED_EXT* texture environment function allows us to perform $2\left(\frac{1}{2}r(a) + \frac{1}{2}r(b)\right) - \frac{1}{2} = r(a) + r(b) - \frac{1}{2}$ in a single pass.

Table 3. Correspondence of operations in numbers and intensities. *Numbers* refer to the real values for which a computation should take place, whereas *intensities* refer to the encoded representations of these numbers in the graphics internal number formats.

Numbers	**Intensities**
$\longrightarrow \quad r : x \to \frac{1}{2\rho}(x + \rho) \quad \longrightarrow$	
$a \in [-\rho, \rho]$	$r(a) \in [0, 1]$
$a + b$	$2\left(\frac{1}{2}r(a) + \frac{1}{2}r(b)\right) - \frac{1}{2}$
ab	$\frac{1+\rho}{2} - \rho(r(a)(1-r(b)) + r(b)(1-r(a)))$
$\alpha a + \beta$	$\alpha r(a) + (\frac{\beta}{2\rho} + \frac{1-\alpha}{2})$
$\max(a, b)$	$\max(r(a), r(b))$
$f(a_0, \ldots, a_n)$	$(r \circ f \circ r^{-1})(r(a_0), \ldots, r(a_n))$
$\sum_\alpha \alpha a_\alpha$	$\sum_\alpha \alpha r(a_\alpha) + \frac{1}{2}(1 - \sum_\alpha \alpha)$
$\longleftarrow \quad \rho(2y - 1) \leftarrow y : r^{-1} \quad \longleftarrow$	

A more universally applicable extension is *NV_register_combiners*, because between the combiners intermediate values can range in $[-1, 1]$ without encoding. So not only addition, but even the linear combination and the multiplication require only one pass, if the scaling and biasing factors are small enough.

Sometimes, not only single rendering passes but entire computations could be saved, if, for example, we knew that we have already approximated the solution of our problem up to the given precision, so that any further calculations would not lead to a better result. The idea would be to calculate the error vector in graphics hardware and then examine its values, for example by computing a vector norm of it. But reading an entire image from the graphics memory to the main memory for this purpose is, in comparison to internal graphics operations, a very slow process. Instead, the histogram extension offers the possibility to obtain a histogram of pixel intensities for an image, requiring to transfer far less data. Given such an histogram $H : \{0, \ldots, 255\} \to \mathbb{N}$ which assigns the number of appearances to every intensity of an image $\bar{V}$, the different vector norms with coefficient exponents $k = 1, 2, \ldots$ can be computed by $\|\bar{V}\|_k = \left(\sum_{y=0}^{255} \left(r^{-1}(y)\right)^k \cdot H(y)\right)^{\frac{1}{k}}$, and for $k = \infty$ we simply pick up the largest $|r^{-1}(y)|$ with $H(y) > 0$, where r^{-1} is the inverse transformation from intensities to numbers.

The color matrix offers another way to save rendering passes. Because fast texture copy from the framebuffer requires both to have the same format, we usually have three color-components RGB available for storage and processing. As all color-components are processed in each op-

eration we may perform parallel computations on them. However, to obtain the final result, we need a possibility to merge the different color values, and this is given by the color matrix.

Finally, we should discuss precision issues when dealing with nonlinear functions. Certainly we cannot resolve the result of a nonlinear function better than the fixed precision permits, but in many cases we are bound to do it far worse than that. When evaluating terms of the form $\frac{x}{y}$ or $(x^k + y^k)^{\frac{1}{k}}$ we may obtain very erroreneous results, because we must compute them sequentially. So for sufficiently small numbers y the application of the inverse function will result in a very large number, which cannot be encoded in $[0, 1]$ anymore; and so even if $x = y > 0$ and thus $\frac{x}{y} = 1$ we will obtain x, since $\frac{1}{y}$ will have been clamped to 1. In such cases the application of a function of several variables is recommended. The lower half of Table 2 lists the relevant functions.

4. The Matrix Vector Product

One of the most common linear algebra operations, required especially in our Finite Element discretizations, is the matrix vector product. We therefore must explain how it can be realized in graphics operations.

Our aim is to express the product of a matrix with s nonvanishing bands and a vector, in terms of a short series of vector operations, of which we have already seen that they can be implemented in graphics hardware. We examine the band matrices, because they represent the typical matrix form occuring in Finite Element discretizations

We will use α, β as indices for matrices and vectors and $\gamma = \beta - \alpha$ as an index offset. The general reformulation makes no reference to the vector image correspondence of Section 3.1, so first we may think of α, β as 'normal' indices.

We are given a matrix $\bar{\bar{A}} = (\bar{\bar{A}}_{\alpha,\beta})_{\alpha,\beta}$ and a vector $\bar{X} = (\bar{X}_\alpha)_\alpha$ and are interested in the resulting vector of the matrix vector product $(\bar{\bar{A}}\bar{X})_\alpha = \sum_\beta \bar{\bar{A}}_{\alpha,\beta}\bar{X}_\beta$. The subdiagonals of $\bar{\bar{A}}$ are given by $\bar{A}^\gamma := (\bar{\bar{A}}_{\alpha-\gamma,\alpha})_\alpha$ and are vectors. Let Γ_A be the set of the γ-indices corresponding to the s nontrivial subdiagonals of $\bar{\bar{A}}$. Moreover, we define the index shift operators $T_\gamma(\bar{V}) := (\bar{V}_{\alpha-\gamma})_\alpha$. The index differences in the above definitions may evaluate to indices outside of the index range of the matrix or vector. For this cases we define the value of the matrix or vector to be zero. Now we may reformulate the matrix vector product:

$$\begin{aligned}(\bar{\bar{A}}\bar{X})_\alpha &= \sum_\beta \bar{\bar{A}}_{\alpha,\beta}\bar{X}_\beta = \sum_{\gamma\in\Gamma_A} (\bar{A}^\gamma)_{\alpha+\gamma}\bar{X}_{\alpha+\gamma}\,, \\ \bar{\bar{A}}\bar{X} &= \sum_{\gamma\in\Gamma_A} T_{-\gamma}\left(\bar{A}^\gamma \bullet \bar{X}\right). \end{aligned} \tag{1}$$

In Finite Element discretizations of partial differential equations the number of nontrivial subdiagonals $s = |\Gamma_A|$ is fairly small, so that there are only few subdiagonals to be stored and the above sum can quickly be evaluated.

For the implementation let us recall from Section 3.1 that our vectors $\bar{X} = (\bar{X}_\alpha)_\alpha$ are enumerated by 2-dimensional indices $\alpha \in (0, \ldots, n-1) \times (0, \ldots, n-1)$ (cf. Fig. 2). Thus a matrix in our context $\bar{\bar{A}}$ is defined by the n^4 values $(\bar{\bar{A}}_{\alpha,\beta})_{\alpha,\beta}$. The popular perception of a matrix as a 2-dimensional number agglomeration may lead here to some confusion. In fact, we would need a 4-dimensional texture for an equivalent representation of a full matrix. But as already indicated, we we will only need to store few subdiagonals which are vectors and thus represented by images. As we already know how to perform addition and multiplication on vectors, in view of (1) we only need to say which graphics operation corresponds to the index shifts T_γ for $\gamma \in \{(-(n-1), \ldots, n-1) \times (-(n-1), \ldots, n-1)\}$. From Section 3.1 we recall that the index of a vector component corresponds to the x, y pixel position in its image representation. Thus an index shift by $\gamma = (\gamma_x, \gamma_y)$ corresponds to an image shift by γ_x pixels in x-direction and γ_y pixels in y-direction. This can be simply accomplished by drawing a (γ_x, γ_y)-shifted copy of the image into the framebuffer or by offsetting the texture coordinates by $(-\gamma_x, -\gamma_y)$, while accessing it from the graphics memory.

We may summarize the matrix vector product as follows. The nontrivial subdiagonals of the matrix in our numerical scheme are computed from the initial data and stored in textures. When a matrix vector product is required, the textures representing the subdiagonals are subsequently multiplied with the texture representing the vector, then shifted and added. The resulting sum is the representation of the resulting vector from the matrix vector product.

5. Solving a Linear System of Equations

With the availability of a matrix vector product realizable in graphics hardware we can now implement an iterative solver for a linear system of equations. This is the core component of most Finite Element codes.

We are given a sparse linear system of equations

$$\bar{\bar{A}}\bar{U} = \bar{R}, \tag{2}$$

with the matrix $\bar{\bar{A}} \in \mathbb{R}^{n,n}$ and the right hand side vector $\bar{R} \in \mathbb{R}^n$ and want to obtain the vector $\bar{U} \in \mathbb{R}^n$ approximating the exact solution, by applying an iterative solver: $\bar{X}^{l+1} = F(\bar{X}^l), \quad \bar{X}^0 = \bar{R}$. Typical solvers are the the Jacobi iteration

$$F(\bar{X}) = \bar{\bar{D}}^{-1}(\bar{R} - (\bar{\bar{A}} - \bar{\bar{D}})\bar{X}), \qquad \bar{\bar{D}} := \mathrm{diag}(\bar{\bar{A}})$$

and the conjugate gradient iteration

$$\begin{aligned} F(\bar{X}^l) &= \bar{X}^l + \frac{\bar{r}^l \cdot \bar{p}^l}{\bar{\bar{A}}\bar{p}^l \cdot \bar{p}^l}\bar{p}^l, \\ \bar{p}^l &= \bar{r}^l + \frac{\bar{r}^l \cdot \bar{r}^l}{\bar{r}^{l-1} \cdot \bar{r}^{l-1}}\bar{p}^{l-1}, \quad \bar{r}^l = \bar{R} - \bar{\bar{A}}\bar{X}^l. \end{aligned}$$

We see that all the operations needed are available in graphics hardware (cf. Table 1, Equation 1). The so far unmentioned scalar product of two vectors can be rewritten as $\bar{V} \cdot \bar{W} = \|\bar{V} \bullet \bar{W}\|_1$, and the inversion of the matrix $\bar{\bar{D}}$ is simply the application of a componentwise inverse function to a vector, because $\bar{\bar{D}}$ is a diagonal matrix comprising only a nontrivial main diagonal vector. For the Jacobi solver even the smaller set of operations available in the texel based implementation suffices (Table 2 upper half). Of course, during the calculations we must always observe the encoding of numbers, by replacing any operations on numbers by the transformation formulas for operations on image intensities given in Table 3.

6. Linear Heat Equation

In this section we present a graphics hardware solver for the linear heat equation. The discussion of this well known partial differential equation will help us to understand the more complex model of the anisotropic diffusion derived from it in the next Section 7..

We consider the time dependent temperature distribution $u : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ in the domain $\Omega := [0,1]^2$. The initial temperature function u_0 at the point in time $t = 0$ and static heaters respectively coolers in form of the function $f : \Omega \to \mathbb{R}$ are given. For simplicity, let the temperature of the borders be constantly zero. Then the evolution of the temperature distribution u is governed by the linear heat equation:

$$\begin{aligned} \partial_t u - \Delta u &= f \ , \quad \text{in } \mathbb{R}^+ \times \Omega \, , \\ u(0,\cdot) &= u_0 \, , \quad \text{on } \Omega \, . \end{aligned} \tag{3}$$

We discretize the domain Ω with an equidistant $n \times n$ grid, and the analytical functions over Ω with discrete functions represented by nodal vectors consisting of the values of the analytical functions at the grid nodes. These nodal vectors are represented in graphics hardware by images of equivalent size. A detailed description of the correspondence between the nodal vectors and the representing images is given in Section 3.1. Furthermore we discretize the time parameter t into an ascending series of points in time $t_0 = 0 < t_1 < t_2 < \ldots$. For each point in time k, there is a corresponding solution vector $\bar{U}^k$ to be found, apart from the first vector $\bar{U}^0$which is given as initial data. The following linear system of equations allows us to compute the next solution vector $\bar{U}^{k+1}$ for the point in time $k+1$ from the current solution vector $\bar{U}^k$

$$\begin{aligned} \underbrace{\left(\bar{\bar{I}} + \frac{\tau_k}{h^2}\bar{\bar{L}}\right)}\, \bar{U}^{k+1} &= \underbrace{\bar{U}^k + \tau_k \bar{F}} \\ \bar{\bar{A}} \qquad \bar{U}^{k+1} &= \bar{R}(\bar{U}^k), \end{aligned}$$

where $\tau_k = t_{k+1} - t_k$ is the current timestep width, $h = \frac{1}{n-1}$ the grid specific diameter, $\bar{F}$ the nodal vector corresponding to the given function f, $\bar{\bar{I}}$ the identity matrix, and $\bar{\bar{L}}$ the stiffness matrix related to the Laplacian. Strictly speaking, we have discretized the problem with bilinear conforming Finite Elements on an equidistant quadrilateral $n \times n$ grid and an implicit first order Euler time scheme, using a lumped mass matrix [15].

The stiffness matrix $\bar{\bar{L}}$ in this case has a fairly simple structure, because all of its subdiagonals are constant. The values of these constants arise from the specific Finite Element discretization we use. The constant values of the main diagonal and the 8 subdiagonals are arranged in the following stencil:

$$S_L = \begin{pmatrix} -\frac{1}{3} & -\frac{1}{3} & -\frac{1}{3} \\ -\frac{1}{3} & \frac{8}{3} & -\frac{1}{3} \\ -\frac{1}{3} & -\frac{1}{3} & -\frac{1}{3} \end{pmatrix}.$$

The α-component $(\bar{\bar{L}}\bar{X})_\alpha$ of the resulting vector $(\bar{\bar{L}}\bar{X})$ in a matrix vector product thus is the sum of $\frac{8}{3}$ times $\bar{X}_\alpha$ plus $-\frac{1}{3}$ times all the values of $\bar{X}$ at the neighboring nodes of α. Therefore $\bar{\bar{L}}\bar{X}$ is nothing else than a convolution of the image representing $\bar{X}$ with the stencil S_L.

Although at first it may seem that we have departed far away from an implementation in graphics hardware talking solely about vector components and operations, we have in fact almost specified the concrete implementation for the linear heat equation.

If we look closely, we see that by knowing the subdiagonals of $\bar{\bar{L}}$, we also know those of $\bar{\bar{A}} = \bar{\bar{I}} + \frac{\tau_k}{h^2}\bar{\bar{L}}$, and with $\bar{F}$ being the nodal vector of the user defined function f we can easily identify the right hand side vector $\bar{R} = \bar{U}^k + \tau_k \bar{F}$. Hence, we have a defined linear system of equations just like in Section 5., where we have shown to be able to solve it in graphics hardware. Below we have summarized the whole process in pseudo code notation:

```
linear heat equation {
  load the images related to u_0, f and the parameters n, τ_k;
  encode the images in graphics memory Ū^0, F̄;
  (From now on perform all operations on image intensities
  according to the transformation formulas from Table 3.)
  for each timestep k {
    store the right hand side image R̄^k = Ū^k + τ_k F̄;
    initialize the iterative solver X̄^0 = R̄^k;
    for each iteration l
      calculate a step of the iterative solver X̄^{l+1} = F(X̄^l);
      (Each time a matrix vector product is needed, apply
      the subdiagonal reformulation (1) to compute it.)
    store the solution Ū^{k+1} = X̄^{l+1}
  }
}
```

7. Anisotropic Diffusion in Image Processing

In this section we present a graphics based acceleration of the anisotropic diffusion model in image processing, which is a full grown application used for advanced edge sensitive denoising of images.

The nonlinear diffusion models were first introduced by a work of Perona and Malik [13], who created a model that allows for denoising of images while retaining and enhancing edges. The regularized model was derived by Cattè et. al. [1] and Weickert [17] introduced an anisotropy depending on the so called structure tensor of images, that steers a nonlinear diffusion process taking care of tangential and normal directions on edges. Concerning the numerical implementation Kačur and Mikula [9] suggested a semi-implicit Finite Element implementation for the isotropic diffusion, on which our presentation here is based.

We consider the unknown $u : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ in the domain $\Omega := [0,1]^2$. An initial noisy image as a function u_0 at the point in time $t = 0$ and a contrast enhancing function $f : \mathbb{R} \to \mathbb{R}$ which depends on u are given. The idea is to evolve the initial image through a partial differential equation such that in the multiscale of the resulting images $u(t,.), t > 0$ the noise dissolves and the contrast and edges enhance with progressing time. This may be compared to the evolution of the temperature distribution in the last section, where starting with an inhomogeneous initial distribution u_0, we obtain a multiscale of temperature functions $u(t,.), t > 0$ in which the initial unevenness levels out in time. However, here we do not use the Laplacian Δ, which is responsible for the homogeneous diffusion of the temperature, but the modified term $\operatorname{div}\left(B^T g(\nabla u_\epsilon) B \nabla u\right)$, which steers the diffusion both in force and direction depending on ∇u_ϵ, a mollification of the gradient ∇u, for example through the convolution with a Gaussian kernel. Thus we can detect edges, diffuse them in the tangential direction and protect them from diffusion in the normal direction. The 2×2 matrices B and g are defined by

$$B(\nabla u) := \begin{pmatrix} \partial_x u & \partial_y u \\ -\partial_y u & \partial_x u \end{pmatrix}, \tag{4}$$

$$g(\|\nabla u_\epsilon\|) := \begin{pmatrix} g_1(\|\nabla u_\epsilon\|) & 0 \\ 0 & g_2(\|\nabla u_\epsilon\|) \end{pmatrix},$$

where typical functions in g are $g_1(x) = 1$ and $g_2(x) = \frac{1}{1+c_g x^2}$. Hence the modified partial differential equation has the following form:

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

We use again the bilinear Finite Elements and deviate from the linear heat equation discretization mainly in the time scheme, where we now use a semi-implicit scheme, evaluating the linear ∇u at the current timestep and the nonlinear $B^T g(\nabla u_\epsilon) B$ and $f(u)$ explicitly at the previous timestep. Hence, we formally obtain the same linear system of equations

$$\left(\bar{\bar{I}} + \frac{\tau_k}{h^2}\bar{\bar{L}}(\bar{U}^k)\right)\bar{U}^{k+1} = \bar{U}^k + \tau_k \bar{F}(\bar{U}^k) \tag{5}$$

with the current timestep width $\tau_k = t_{k+1} - t_k$ and the grid specific diameter $h = \frac{1}{n-1}$, but with a different stiffness matrix $\bar{\bar{L}}$, which strongly depends on $\bar{U}^k$.

Here, the components of $\bar{\bar{L}}(\bar{U}^k)$ vary globally with their index, i.e. the representing subdiagonal images have different values across all its pixels, unlike the stiffness matrix of the linear heat equation (4) where all pixels had the same value. Therefore, we can specify the subdiagonals $\bar{L}^\gamma$ only locally, in dependence on the local $\bar{U}^k$ values. The key point to remember is, that the stiffness matrix $\bar{\bar{L}}(\bar{U}^k)$ inherits the property of locally steering the diffusion in force and direction, from the weight matrix

$$G := B^T g(\nabla u_\epsilon) B. \tag{6}$$

To specify $\bar{\bar{L}}$ first we have to evaluate this 2×2 weight matrix G. This is accomplished by substituting the discrete gradient ∇U^k, evaluated at the center of each grid-element E (cf. Fig. 2), for the analytical gradient ∇u in the definitions (4). Thus we obtain the discrete weight matrix G_E^k for each element of the grid.

With G_E^k we can identify the subdiagonals $\bar{L}^\gamma$ of the stiffness matrix as

$$\bar{L}_\alpha^\gamma = \sum_{E \in E(\alpha)} \sum_{i,j \in \{x,y\}} (G_E^k)_{i,j} (S^\gamma)_{i,j}, \tag{7}$$

where $E(\alpha)$ is defined as the set of the 4 elements around the node α, the indices $i, j \in \{x, y\}$ address the 2×2 matrices and

$$(S^\gamma)_{i,j} := \int_{[-1,1]^2} \partial_i \hat{\Phi}(x,y)_{(0,0)} \cdot \partial_j \hat{\Phi}(x,y)_\gamma \, \mathrm{d}x \, \mathrm{d}y$$

are factors which can be precomputed and depend on specific Finite Element basis functions. We can analyze the whole term (7) as follows: G_E^k contains the right weight distribution to steer the diffusion along the edges but not across them; S^γ contains the discretization dependent factors with which to assign the weights to the nodes; the sum $\sum_{i,j \in \{x,y\}}$ summarizes the effects calculated separately for the direction combinations and the sum $\sum_{E \in E(\alpha)}$ lets all the neighboring elements contribute to the entry of this node.

For the implementation we must explain how we can build up the right hand side vector $\bar{F}(\bar{U}^k)$ and the subdiagonal vectors $\bar{L}^\gamma(\bar{U}^k)$ of the stiffness matrix from the vector $\bar{U}^k$ in graphics hardware.

A typical choice for f is $f(x) = c_f |u^0(x) - u^k(x)|$, from which we deduce the nodal vector $\bar{F}(\bar{U}^k) = c_f |\bar{U}^0 - \bar{U}^k|$, with $|.|$ operating on components. Obviously, we can easily implement this in graphics hardware. For the construction of $\bar{L}^\gamma(\bar{U}^k)$ we must calculate the values $(G_E^k)_{i,j}$ and $(S^\gamma)_{i,j}$ for all grid-elements E, indices $i, j \in \{x, y\}$ and offsets $\gamma \in \{0, \pm 1\} \times \{0, \pm 1\}$ (cf. 7).

As $(S^\gamma)_{i,j}$ do not depend on the grid-element, they can be precomputed as $4 * 9 = 38$ different values. The weight values $(G_E^k)_{i,j}$ are more difficult to handle. We organize

them in 4 images $G^k_{(i,j)} := (G^k_{E,(i,j)})_E$, each containing the values on all elements with common (i,j)-index.
Before invoking the solver for the linear equation system, we compute the images $\partial_x \bar{U}^k := (\partial_{x,E} \bar{U}^k)_E$ and $\partial_x \bar{U}^k := (\partial_{y,E} \bar{U}^k)_E$ by differencing the values of $\bar{U}^k$ on the nodes of each element in the x and y direction respectively. From these we compute the image $\|\nabla U^k\| = \sqrt{(\partial_x \bar{U}^k)^2 + (\partial_y \bar{U}^k)^2}$ and then convolute it with a Gaussian kernel, represented by a stencil, to $\|\nabla U^k_\epsilon\|$. Finally, we evaluate the function g_2 to obtain the image $g_2(\|\nabla U^k_\epsilon\|)$.
Then we have all the discrete equivalents of the entries of the matrices B and g (4) by which G has been defined (6). Hence we obtain the 4 images $G^k_{(i,j)}$ as the 4 entries resulting from the computation $B^T(\nabla U^k) g(\|\nabla U^k_\epsilon\|) B(\nabla U^k)$.

Figure 3. One step of the anisotropic diffusion model apllied to noisy 128^2 images. The computation in graphics hardware takes about 0.05s ensuring real time performance.

The examples in Fig. 3 underline the edge conserving property of the anisotropic diffusion model. Performance issues are discussed in the following Section 8..

8. Performance Measurements and Conclusions

All computations have been performed on an ELSA Gladiac Ultra graphics card powered by NVIDIA's GeForce2 Ultra chip. We have used the texture environment based implementation (Table 2) with the NV_register_combiners extension (cf. Section 3.4) and RGB8 textures under the $[-1,1] \to [0,1]$ encoding representing 128^2 vectors. All matrix vector products have been computed by applying the subdiagonal reformulation (1), even where a convolution would have sufficed, and the Jacobi solver with the fixed number of 10 iterations has been used for solving of the linear systems of equations.
In the case of the linear heat equation one iteration of the Jacobi solver takes approximately 1.6ms. This equivalents more than 300 MOP/s, a value hardly reachable by pure software implementations on nowadays PCs. The same applies to the anisotropic diffusion. Here one iteration of the Jacobi solver took approximately 5ms, thus computing the images of Fig. 3 in about 0.05s.
Besides these promising results, there are however some important issues to discuss. We have been using a very restricted set out of the introduced operations. This is because, only these operations actually exploit the main advantage of the graphics hardware, namely the superior memory bandwidth. Activating an operation which does not do that, hits performance by a huge factor. Therefore we had to approximate all involved nonlinear functions by linear in the implementation of the anisotropic diffusion. This leads to an deterioration in image quality in the following timesteps. The feedback function *glHistogram* which enables adaptive iteration abort is also too slow to be used. Moreover, larger number intervals than $[-1,1]$, cost many more passes, because unlike the slow *glPixelTransfer*, scaling and biasing with intensities $|a| > 1$ is very restricted. Finally, the restricted precision of 8 bits per color component leads to unsatisfying results for the linear heat equation, because smooth transitions in temperature produce very small values in the convolution, with very high relative errors.
But the remaining restrictions do not distract us from the looming possibilities. Therefore, we want to consider here a few graphics hardware developments which would be very beneficial to numerical implementations. Some of these are already on the way, others already strongly advocated by the graphics community, but let us list them all to point out the needs:

- Numbers
  - Signed textures, signed color buffer values.
  - Exact representation of $-1, 0, 1$ in the fixed point number format (very important for iterations).
  - Cumulating high precision formats like LA16 or L32 in addition to RGBA8.
  - Fast scale, bias out of $(-\infty, \infty)$.
- Operations
  - Direct rendering to arbitrary textures.
  - Fast dependent texture lookup.
  - 3D texture hardware support (greatly simplifies computations on 3D data).
  - Volumetric rendering to 3D textures.

Obviously, this list is by no means complete. Instead, we have concentrated on graphics features within reach of the forthcoming GPU generation and hope that they can be a starting point for further considerations of graphics hardware applications in scientifc computations.

References

[1] F. Catté, P.-L. Lions, J.-M. Morel, and T. Coll. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM J. Numer. Anal.*, 29(1):182–193, 1992.

[2] C.C. Douglas, J. Hu, M. Lowarschik, U. Rüde, and C. Weiß. Cache optimization for structured and unstructured multigrid. *Electronic Transactions on Numerical Analysis (ETNA)*, 1999.

[3] W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. In *ACM Symposium on Interactive 3D Graphics*. ACM/Siggraph, 1999.

[4] H. Hellwagner, U. Rüde, L. Stals, and Chr. Weiß. Data locality optimizations to improve the efficiency of multigrid methods. In *Proc. 14th GAMM Seminar 'Concepts of Numerical Software'*, Kiel, 1998. Vieweg.

[5] U. Hoffmann, M. Meißner, and W. Straßer. Enabling classsification and shading for 3d texture mapping based volume rendering using opengl and extensions. In *Proc. Visualization '99*, pages 207–214, 1999.

[6] M. Hopf and T. Ertl. Accelerating 3d convolution using graphics hardware. In *Proc. Visualization '99*, pages 471–474. IEEE, 1999.

[7] M. Hopf and T. Ertl. Accelerating Morphological Analysis with Graphics Hardware. In *Workshop on Vision, Modelling, and Visualization VMV '00*, pages 337–345, 2000.

[8] B. Jobard, G. Erlebacher, and M. Yousuff Hussaini. Hardware-accelerated texture advection for unsteady flow visualization. In *Visualization '00*, pages 155–162, 2000.

[9] J. Kačur and K. Mikula. Solution of nonlinear diffusion appearing in image smoothing and edge detection. *Appl. Numer. Math.*, 17 (1):47–59, 1995.

[10] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings IEEE Visualization '99*, pages 355–362, 1999.

[11] L. Lippert. *Wavelet-based Volume Rendering*. PhD thesis, Department of Computer Science, ETH Zrich, 1998.

[12] OpenGL Architectural Review Board (ARB), http://www.opengl.org/. *OpenGL: graphics application programming interface (API)*, 1992.

[13] P. Perona and J. Malik. Scale space and edge detection using anisotropic diffusion. In *IEEE Computer Society Workshop on Computer Vision*, 1987.

[14] M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 75–84. Springer, 2001.

[15] V. Thomee. *Galerkin - Finite Element Methods for Parabolic Problems*. Springer, 1984.

[16] S. Turek. Some basic concepts of feast. In *Proc. 14th GAMM Seminar 'Concepts of Numerical Software'*, Kiel, 1998. Vieweg.

[17] J. Weickert. Theoretical foundations of anisotropic diffusion in image processing. *Computing*, Suppl. 11:221–236, 1996.

[18] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. *Computer Graphics (SIGGRAPH '98)*, 32(4):169–179, 1998.