

Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages

Ivan Viola

Armin Kanitsar

Meister Eduard Gröller*

Institute of Computer Graphics and Algorithms
Vienna University of Technology, Austria

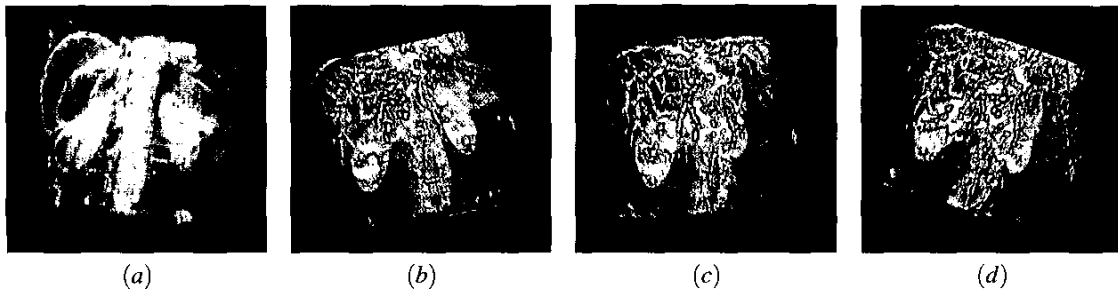


Figure 1: Liver dataset (a) and its segmented vessel structure using median (b), bilateral (c) and rotated mask filters (d).

Abstract

Non-linear filtering is an important task for volume analysis. This paper presents hardware-based implementations of various non-linear filters for volume smoothing with edge preservation. The Cg high-level shading language is used in combination with latest PC consumer graphics hardware. Filtering is divided into per-vertex and per-fragment stages. In both stages we propose techniques to increase the filtering performance. The vertex program pre-computes texture coordinates in order to address all contributing input samples of the operator mask. Thus additional computations are avoided in the fragment program. The presented fragment programs preserve cache coherence, exploit 4D vector arithmetic, and internal fixed point arithmetic to increase performance. We show the applicability of non-linear filters as part of a GPU-based segmentation pipeline. The resulting binary mask is compressed and decompressed in the graphics memory on-the-fly.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.4.3 [Image Processing and Computer Vision]: Enhancement—Filtering

Keywords: Non-linear Filtering, Segmentation, Hardware Acceleration

*{viola|kanitsar|meister}@cg.tuwien.ac.at,
<http://www.cg.tuwien.ac.at/home/>

IEEE Visualization 2003,
October 19-24, 2003, Seattle, Washington, USA
0-7803-8120-3/03/\$17.00 ©2003 IEEE

1 Introduction

Medical datasets obtained from CT or MRI scanners contain noise. It is due to scanner precision, movement artifacts, inhomogeneous contrast-agent distribution and many other factors. All these factors strongly influence the correct reconstruction as well as the diagnostic process. To improve noisy data, filtering turns out to be a fundamental task for feature enhancement, data analysis, noise removal, and finally reconstruction.

Depending on a filter type, filtering is either linear or non-linear. Linear filtering is a convolution of the dataset with a filter kernel, which is given as a continuous function or through discrete samples. The continuous filters respectively their discrete high resolution representations are typically used for reconstruction purposes. Another class are low-resolution filters adopted from popular linear image processing operators. These are a very rough representations of the continuous kernels. However they are often sufficient for smoothing, edge detection or gradient estimation. Typical representatives are mean or Gaussian smoothing operators and edge detectors like Sobel or Laplacian operators [Sonka et al. 1995].

Non-linear filters are generally filters that do not fit into the category mentioned above, i.e., the filtering cannot be expressed as a convolution. Typical examples are dilation, erosion, and median filters used for volume analysis. In this paper we focus on a subcategory of non-linear filters, i.e., edge-preserving smoothing filters. The advantage as compared to linear smoothing operators is that they smooth areas within a particular object, while preserving its borders.

Edge-preserving smoothing is often used as a preprocessing step for medical image segmentation. It generally improves the accuracy of the segmentation by preserving object boundaries, while reducing random noise in the interiors of the structures. However, such filters require more complex processing than linear operators, so the filtering performance is in most cases worse. We adapt these filters to the latest consumer graphics hardware. The segmentation via thresholding is also done on the graphics hardware with the pos-

sibility to interactively adjust the threshold value. The final mask is then stored in a simple compressed form in a stack of pbuffers. This compact representation can be transferred back to the main memory much faster than a mask with the size of the original dataset. The compact representation can be used for permanent storage of the segmentation mask. When the mask is used for volumetric clipping with volume textures directly on hardware [Weiskopf et al. 2002], it can be decompressed on-the-fly without significant performance loss. The filtering and segmentation pipeline is illustrated in figure 2.

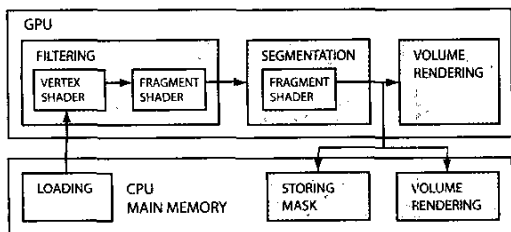


Figure 2: The filtering and segmentation pipeline.

The contribution of this paper is to move the data enhancement step of the visualization pipeline to the graphics hardware. Currently the graphics hardware is used mostly for the last step, i.e., rendering. However the performance and flexibility of the latest graphics hardware can be used for other steps as well. The idea is to upload the dataset into the graphics hardware as textures and perform all possible tasks on the GPU. We add the pre-filtering step using complex non-linear filters and the segmentation step. The filtering is shown on three specific edge-preserving filter types. We also point out general recommendations for hardware-based filtering algorithms that target the current as well as future hardware generations.

The remainder of the paper is organized as follows: Section 2 describes features of current graphics hardware. In section 3 we give an overview on hardware-based filtering and non-linear filtering in software. Section 4 describes GPU based filtering in general and then points out specific issues of each filter type implementation. After the filtering stage of the pipeline, the segmentation stage is discussed in section 5. Results are shown in section 6 and conclusions are drawn in section 7.

2 Graphics Hardware Issues

In our implementation we exploit the programmable features of the current graphics hardware such as vertex and fragment processing units. These features make the graphics hardware very flexible and allow to implement even complex filters efficiently. A vertex shader allows many manipulations upon vertices, e.g., changing position, color, texture coordinates or to compute information which is passed to the fragment shader. Vertex shaders currently support conditional jumps and subroutine calls. Fragment programs are similar, but operate on fragments. For both processing units, a small assembler-like program is loaded to the graphics memory and executed for each vertex respectively fragment. Most instructions are four-component vector instructions. For example multiplication of two `float4` registers is a component-wise operation, i.e., four multiplications can be computed within one instruction. The number of instruction slots for a vertex respectively fragment program is limited by the graphics hardware. A fragment program, however, does not support conditional jumps and subroutines, so the number of instruction slots limits the number of instructions executable

per fragment. If the filtering process does not fit into one rendering pass, i.e., the number of required instructions exceeds the hardware limit, the intermediate results can be rendered into pbuffers. A pbuffer (pixel puffer) is an offline rendering context which is not displayed, but has the same features as a framebuffer. Pbuffers are not limited by the actual screen resolution, because the rendering context size is limited by only the graphics hardware. Pbuffers can be directly bound as textures in the next rendering pass without the need to copy within the graphics memory. These buffers can have up to 32-bit floating-point precision per color channel.

Recently various C-like high-level shading languages (HLSL) have been proposed and developed. They enable the compiler to produce assembly code and offer the developer higher programming comfort. Beside this, it is possible to share the same high-level code among different hardware platforms. The high-level code is then compiled to hardware specific assembly code in run-time. The principle is derived from the RenderMan shading language [Upstill 1990], however RenderMan was not intended for real-time rendering. The first HLSL using the programmable graphics hardware is the Stanford Real-Time Shading Language (RTSL) [Proudfoot et al. 2001], which allows programmability for four so called *computation frequencies*: constant, per primitive group, per vertex, and per fragment. Beside the RTSL other HLSLs have been developed like DirectX HLSL [2003], OpenGL Shading Language [2003] or NVIDIA's Cg Language [Mark et al. 2003]. Cg has some additional advantages as compared to the other shading languages. It supports both hardware APIs, i.e., OpenGL as well as DirectX. For easier data transfer Cg uses pre-defined structures called *binding semantics* to be used in the vertex shader and vertex-shader to fragment-shader communication. The API-to-vertex binding semantics contain all vertex attributes like color, normal vector, and texture coordinates, which can be arbitrarily changed in the vertex program. After the transforms have been done, the output is stored in vertex-to-fragment binding semantics. The fragment shader receives the interpolated colors and texture coordinates as input parameters. The input and output binding semantics are denoted as IN for input and OUT for output. The output binding semantics (OUT) of the fragment shader is output fragment color, and eventually depth value. Because of this flexible and simple model we use the Cg Language in our implementation.

3 Related Work

The area of hardware-accelerated filtering has gained wider attention recently. 3D convolution using graphics workstations was introduced by Hopf et al. [1999]. They perform the linear filtering on 2D slices using the imaging subset of OpenGL 1.2. The separability property of specific low pass and high pass kernels is exploited. The filtered 2D slices are then convolved along the third dimension via 1D convolution. Later they extended their work on hardware-based filtering by morphological operators using multi-pass blending with min and max operations [Hopf and Ertl 2000]. The first graphics hardware generation featuring programmable units was used for 2D convolution by James [2001]. This approach was intended for filtering using small kernels, however the filtering result suffered from precision artifacts. A general approach for hardware-based linear filtering was presented by Hadwiger et al. in [2001; 2002]. Their work can be applied to arbitrary linear filter kernels, gaining speed-up from various kernel properties. The main difference to the previous approaches is to use texture instead of color value for filter kernel representation. A particular sample is not convolved with a constant value, but with a kernel *tile*. This allows to filter with high-resolution kernels for, e.g., cubic reconstruction.

Nowadays hardware-accelerated three-dimensional linear filtering can be implemented in a rather straightforward way. Also implementation of morphological analysis is simple, because current

hardware directly supports the MIN and MAX instructions. However the implementation of filters that require complex operations like sorting, or have iterative behavior is still difficult. These types of filters have not been implemented on the graphics hardware yet, but there are already various software implementations available. We focus on three specific types of edge-preserving smoothing filters. The first is the median operator. Huang et al. [1979] present one of the fastest median filtering algorithms. It is based on storing and updating the gray level histogram of pixels within the operator mask. From one filtered pixel to the next, the $m \times n$ operator mask moves only one column. It means, only n values have to be removed and n added to the histogram, so the computational complexity is $O(n)$. Other median estimation algorithms are listed by Press et al. [2002]. The second and nowadays very popular non-linear filter type comprises bilateral filters introduced by Tomasi et al. [1998]. The filter consists of multiplication of two weights, which are different for each contributing sample. The first weight is based on the *geometric* distance between the neighbor and the center pixel. The second weight is based on the *photometric* similarity between these two pixels. A bilateral filter can be used for edge-preserving smoothing. This kind of edge-preserving smoothing was used recently for displaying high-dynamic-range images [Durand and Dorsey 2002] and smoothing of normals [Tasdizen et al. 2002]. A very similar filter was also proposed by Aurich et al. [1995]. They use a filter chain with varying σ parameters instead of one iteration. The third category of edge-preserving filters are filters proposed by Nagao et al. [1979]. The filter computes mean and dispersion for 9 different operator masks. The output value is the mean of the mask with the smallest dispersion. Similar filters based on similar edge-preserving constraints are also called rotated mask smoothing filters [Sonka et al. 1995].

4 Hardware-Based Non-Linear Filtering

Our filtering methods using programmable hardware are in spirit to previous approaches [James 2001]. Exploiting the latest PC graphics hardware features allows to increase the computational efficiency and performance compared to the previous approaches. In this section we introduce various new techniques that significantly increase the filtering performance.

The workflow of our algorithms is as follows: First we download the volumetric data into the graphics-hardware memory. The dataset is stored as 2D textures in contrast to 3D. This preserves the 12-bit precision of medical datasets by using the floating-point formats. These formats are currently available only for 2D textures on the NVidia hardware. The high-precision fixed-point formats are not yet supported.

For illustrative purposes, we consider filtering with a filter size of $5 \times 5 \times 5$ voxels. For each filtered slice we set up a corresponding pbuffer as a destination rendering context. We render a simple quad geometry and bind 5 textures that contain the contributing voxels. The texture-coordinate vertex-attributes are set to fit the whole texture on the quad. This setup is illustrated in figure 3. For each voxel of the dataset we execute the same vertex and fragment program. The filtered output value is rendered into the corresponding pbuffer. In case of iterative filtering, the pbuffer can be bound directly as texture for the second filtering pass. In the following we describe the main filtering part done on the vertex and fragment processing units.

4.1 Per-vertex stage

To access the value of a particular voxel that contributes to the filtered output value, we have to perform a texture lookup into the source texture. Normally it would be necessary to compute addresses from the texture coordinates of the processed output voxel.

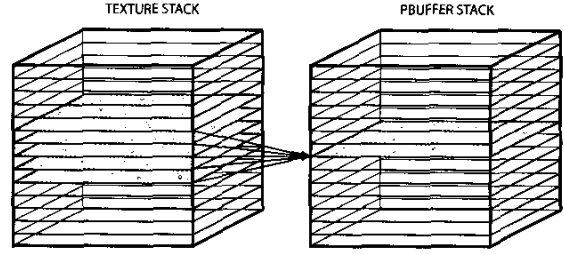


Figure 3: Setup for 2D texture slices and corresponding puffers.

For a $5 \times 5 \times 5$ filter size this would require 125 instructions to compute the addresses of all contributing voxels for each filtered voxel. However, using 2D textures, we have in fact only 25 different addresses within a slice. Current hardware features eight texture coordinates per-vertex. The straightforward way to pre-compute eight addresses to access different locations within the texture, is to set up each texture coordinate with a different offset. These addresses are then interpolated “for free” for each processed fragment. This leads to a higher performance, but it is still necessary to compute the addresses of the remaining voxels. We show how to pre-compute the addresses to all 25 voxels within a slice using just 7 of the 8 available texture coordinates.

Each texture coordinate is a four-dimensional vector. We use the $xyzw$ notation, where x is the first, y is the second, z is the third, and w is the fourth component. The graphics hardware directly uses only xy components for 2D textures. The flexibility of the hardware allows to use zw components for storing addresses as well. This is done with another feature called the *swizzle* operator “.”, which allows the components of a 4D vector register to be arbitrarily rearranged in a new vector. We rearrange the components of a texture coordinate, to generate another texture coordinate without any performance loss. Figure 4 shows a close-up at 5×5 voxels of the rendering context. Three voxels are highlighted to show the correspondence between input and output texture coordinates.

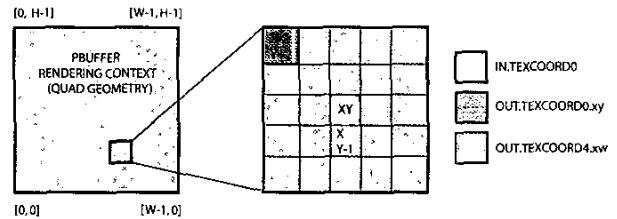


Figure 4: Rendering context of size $W \times H$ and stretched quad with the texture coordinate setup at the vertices (IN.TEXTCOORD0). The close-up shows three highlighted voxels of a 5×5 mask illustrating the correspondence between the input and output texture coordinates (OUT.TEXTCOORD0-6).

Texture coordinates are denoted as TEXTCOORD0-TEXTCOORD7. IN.TEXTCOORD0.xy contains the x and y address of the central voxel. IN.TEXTCOORD0.xyxy for example is a 4D vector which contains as first and third component the x -address, and as second and fourth component the y -address of the central voxel. OUT.TEXTCOORD0 stores in its xy components the address of the neighboring voxel with offset $[-2, 2]$ from the central fragment. The zw components contain the address of the voxel with $[-1, 1]$ offset. Using the swizzle operator we can address also voxels with offsets $[-2, 1]$ (xw components) and $[-1, 2]$ (zy components). In other words one texture coordinate register together with the swizzle operator is used to address four neighboring voxels. Similarly

we can address up to three voxels in a row respectively column. `OUT.TEXCOORD2` stores the `x` and three times the `y` address of the central voxel. `OUT.TEXCOORD6` stores three addresses within one row analogously to the `OUT.TEXCOORD2` binding semantic. Figure 5 shows the complete setup of available texture coordinates to access entire 5×5 neighborhood. Figure 6 gives the corresponding listing of the vertex shader. The overall speed-up achieved by the voxel address pre-computation can be up to a factor of 10, depending on the filter and data type used for address computation.

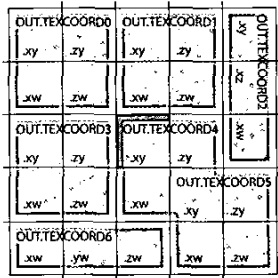


Figure 5: Pre-computed addressing of neighboring samples within one slice.

```

OUT.TEXCOORD0 = IN.TEXCOORD0.xyxy + float4(-2, 2, -1, 1);
OUT.TEXCOORD1 = IN.TEXCOORD0.xyxy + float4( 0, 2,  1, 1);
OUT.TEXCOORD2 = IN.TEXCOORD0.xyyy + float4( 2, 2,  1, 0);
OUT.TEXCOORD3 = IN.TEXCOORD0.xyxy + float4(-2, 0, -1, -1);
OUT.TEXCOORD4 = IN.TEXCOORD0.xyxy + float4( 0, 0,  1, -1);
OUT.TEXCOORD5 = IN.TEXCOORD0.xyxy + float4( 1, -1,  2, -2);
OUT.TEXCOORD6 = IN.TEXCOORD0.xxyy + float4(-2, -1,  0, -2);

```

Figure 6: Texture coordinate setup for the `OUT.TEXCOORD0-OUT.TEXCOORD6` binding semantics.

Using larger operator masks would require more available texture coordinate binding semantics. For example a 6×6 operator mask would require 9 texture coordinates. We can replace the missing texture coordinate by the `COLOR0` binding semantic that is interpolated for each fragment in the same way as the texture coordinates. Also other vertex attributes can be used for addressing the contributing neighborhood.

4.2 Per-fragment stage

Apart from the contributing voxels' address pre-computation the rest of the filtering process is done in the per-fragment stage. Before we describe each particular filter type in detail, we point out some general recommendations, which are important for efficient hardware implementation.

To efficiently access the graphics hardware memory via texture fetches, it is important to preserve cache coherence. A texture lookup operation does not fetch only one texel value, but a whole block of graphics memory, which is then stored in the cache. It is important to know how the texture is stored in the graphics memory, which is different for each texture format. Some formats store the texture linearly. In this case the cache line is loaded with the `x`-axis neighbors. Other formats load a rectangular `xy` block in a cache line. Due to hardware limitations we use linearly stored texture formats. All the texture fetches of contributing neighbors are

performed in the `x`-axis direction. Beside the 2D slices of the volume dataset, we also store pre-computed complex functions in the lookup textures instead of computing them on-the-fly. Such lookup textures usually suffer from cache misses that may trash the cache, because of accessing different parts of the texture. If possible it is recommended to use rather small lookup textures to reduce cache misses.

Another approach to speed-up the process is to take advantage of 4D vector instructions. A lot of instructions perform per-channel operations, even instructions like `MIN` perform component-wise comparisons. This allows to fold up to four instructions into a single one. Storing four scalar values in a 4D vector costs four `MOV` instructions. Therefore the number of instructions is reduced only if the vector instructions are executed more than once per contributing fragment. To eliminate the `MOV` instructions it is necessary to fetch four values within a single texture-fetch instruction, which requires four-channel textures.

The shaders allow floating- as well as fixed-point arithmetics. There are two floating-point possibilities: 32- and 16-bit floating-point precision. Fixed-point arithmetics operates on a 12-bit fixed-point type, which is in the interval $[-2, 2]$. The fixed-point arithmetics is twice as fast as low-precision floating-point arithmetics, which in turn is twice as fast as high-precision floating-point arithmetics. Medical datasets are usually stored in a 16-bit integral type, using only 12 bits. This allows us to use the fixed-point arithmetics to get the maximum performance without precision loss.

Interestingly also the number of used registers strongly influences the overall performance of shader execution. An effective solution could be to rearrange the code into multiple passes instead of a one pass approach.

Another implementation issue that significantly influences the performance is the number of conditional operations in fragment programs. This should be minimized in the HLSL code, because efficient branching is not yet available and all the instructions in both branches are executed respectively. Also the `discard` operation, i.e., early fragment exit, is still not well supported. Instead of the early termination of program execution for the current fragment, all remaining instructions are executed and the output value is just ignored.

If possible the implementation should use the HLSL standard library functions instead of user written routines. In most cases the built-in functions are compiled more efficiently into assembly code.

The hardware implementation should take into account all the mentioned issues, otherwise the computational resources will not be utilized efficiently. The following subsections describe the efficient fragment shader implementation of three different non-linear filters.

4.2.1 Median Filter

The most typical representative of non-linear filters is the median filter. It produces similar results as the mean or Gaussian smoothing in regions of constant color, but preserves edges in image regions, where the pixel intensity rapidly changes. For a random variable x the median M is the value for which the probability of the outcome $x < M$ is equal to 0.5. In image processing the median is the central value of the ordered set of the values within the operator mask. There has been a lot of research involved in efficient median filter implementation. We have already discussed software implementations in section 3. Due to graphics hardware limitations our implementation is not based on such a histogram based approach as proposed by Huang et al. [1979].

First we describe the basic idea of the algorithm and then show how to reduce the number of passes exploiting the 4D arithmetics of current hardware. Our method is based on traversing all voxels within the operator mask, comparing the estimated *pivot* value to all

voxel values within the operator mask \mathbf{Q} . For the $5 \times 5 \times 5$ filter size, the number N of contributing voxels is equal to 125. Because of the 12-bit precision voxel values are in the range of $[0..4095]$. The algorithm performs a binary search on the voxel values within the operator mask. Initially we set the possible range of median values to 0 for the lower border, and 4095 for the upper border. The pivot is set to the average value of the border values. Then we traverse all voxels and evaluate the number of voxel values f , which are greater than the pivot. If the number of greater values is smaller than $\frac{N+1}{2}$, the median is within the range $[0, pivot]$ otherwise it is in the range $[pivot, 4095]$. According to this condition we update the value of the corresponding border to the pivot value and repeat traversing the voxel values again. The pivot is updated to the average value of the new border values. The number of such iterations is equal to $\log(4096)$, i.e., 12. After the 12th iteration the median value is the only integer value, which is within the interval of borders.

Current hardware makes possible to compare the voxel value with four different values using a 4D vector as pivot instead of scalar value. We generate a 4D vector from the voxel value as well and perform component-wise comparisons. The difference to the algorithm described above is that we do not estimate the pivot as average value from the border values, but divide the actual range uniformly into five intervals. The algorithm can be also considered as a histogram search with four bins, where `counter` contains cumulative histogram. This reduces the number of necessary iterations from 12 to 6. The pseudo code of the GPU-based median filter is listed in figure 7.

```

border = {0, 4095};
mult = {.2, .4, .6, .8};
for (s = 0; s < 6; s++){
    counter = 0;
    pivot = border.xxxx+(border.y-border.x).xxxx*mult;
    for all f(i, j, k) in Q
        if (pivot > f(i, j, k)) counter++;
    tmp.border = border.y;
    border.y = pivot.x;
    if(counter.x < 63) {
        border.x = pivot.x;
        border.y = pivot.y;
    }
    if(counter.y < 63) {
        border.x = pivot.y;
        border.y = pivot.z;
    }
    if(counter.z < 63) {
        border.x = pivot.z;
        border.y = pivot.w;
    }
    if(counter.w < 63) {
        border.x = pivot.w;
        border.y = tmp.border;
    }
}
return floor(border.y);

```

Figure 7: Median filter code listing for a $5 \times 5 \times 5$ operator mask \mathbf{Q} , i.e., $N = 125$.

The GPU-based implementation of the median filter requires a large number of conditionals. We are using the most effective condition, i.e., the standard library function `step(a, x)`, which returns 0 in case $x < a$, otherwise it returns 1. Due to an inefficient implementation of the discard function, we cannot gain a speed-

up from early termination. This will be possible on future graphics hardware, as the filter will run more efficiently. The speed-up comparing to the software implementation of `quick_median` [Press et al. 2002] is a factor of 1.97. The result of a thresholding segmentation with pre-filtering using the median is shown in figure 11.

4.2.2 Bilateral Filter

As mentioned in section 3, bilateral filters became more important over the last years. A bilateral filter has a similar behavior like convolution-based smoothing in regions of more or less constant color. The filter, however, eliminates the contribution of samples whose values differ considerably from the center sample. We use the Gaussian case [Aurich and Weule 1995]. The non-linear Gaussian filter is defined by

$$g[x] = \frac{\sum_{q \in \mathbf{Q}} h[q]f[q]}{\sum_{q \in \mathbf{Q}} h[q]}, \quad (1)$$

where \mathbf{Q} is the operator mask, $h[q]$ the composed filter weight and $f[q]$ the value of voxel q . The filter weight is composed in the following way

$$h[q] = h_g[\|q - x\|]h_p[f(q) - f(x)], \quad (2)$$

which is a multiplication of the *geometric* and *photometric* weight. The term $\|q - x\|$ is the Euclidian distance between the contributing and central voxel. Both weights h_g, h_p are computed by a Gaussian function, each with different input and σ values

$$h_g[t] = e^{-\frac{t^2}{2\sigma_g^2}} \quad h_p[t] = e^{-\frac{t^2}{2\sigma_p^2}}. \quad (3)$$

The implementation uses a two-channel register for accumulation. One channel contains sum of the weights computed by multiplication of the *geometric* and *photometric* weight. The other channel sums the weight multiplied by corresponding neighboring voxel. To avoid evaluation of a complex function, we pre-compute the *geometric* weight coefficients in software and send them to the fragment shader via free binding semantics like, e.g., `COLOR0`, `COLOR1` or `NORMAL`. This can be easily done in software, because for a $5 \times 5 \times 5$ operator mask there are only 10 different *geometric* weights. The *photometric* weight is represented as a 1D lookup texture. A straightforward approach would use a 2D lookup texture, where the $(f(q), f(x))$ texture entry contains the corresponding h_p value. The texture fetch will return the pre-computed complex h_p function value. The texture lookup costs one additional MOV instruction, because the texture coordinates must be stored in one register. However, we use a 1D lookup texture, where the difference between two voxel values is the texture coordinate for the lookup. This eliminates the speed-down caused by cache trashing, which is often the case when using large 2D lookup textures. The number of instructions remains the same, only the MOV instruction is replaced by an ADD instruction. The only remaining problem are negative results of the difference operation. This problem can be easily solved by setting the texture wrap mode to mirror the texture. The final output value is computed as a division between the two channels to normalize the accumulated intensity. The filtering performs efficiently, because there is no conditional code at all. The algorithm of the bilateral filter is described in pseudo code in figure 8.

Figure 11 shows the liver dataset segmented via thresholding with pre-filtering using a bilateral filter. We perform more iterations to increase the smoothing effect. The resulting filtered dataset is shown after 1, 3, and 5 iterations. The GPU-based implementation achieves approximately a speedup factor of 1.52.

```

num = den = 0;
for all f(i, j, k) in Q{
  diff = f(i, j, k) - f(central);
  h.p = tex(photometric.tex, diff);
  h(i, j, k) = h.p * h.g(i, j, k);
  num += h(i, j, k) * f(i, j, k);
  den += h(i, j, k);
}
return num / den;

```

Figure 8: Bilateral filter code listing.

4.2.3 Rotating Mask Filter

The last filter which we will discuss in more detail is the rotating mask filter, similar to the Nagao filter [1979]. The basic principle is to divide the operator mask into several sub-regions and compute mean and dispersion σ^2 for each sub-region separately. The output value is the mean of the sub-region with minimal dispersion. We have implemented a 3D version of this filter, considering six different sub-regions, which are sketched in figure 9.

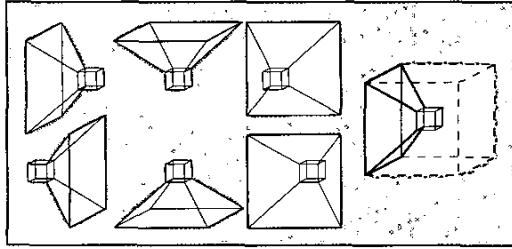


Figure 9: Sub-regions of the decomposed operator mask.

The dispersion is calculated as proposed in [Sonka et al. 1995] by the following equation

$$\sigma^2[S] = \frac{1}{N_S} \left(\sum_{q \in S} f[q]^2 - \frac{\left(\sum_{q \in S} f[q] \right)^2}{N_S} \right), \quad (4)$$

where N_S is the number of contributing voxels per sub-region, S is the sub-region mask and $f[q]$ is the value of voxel q .

The implementation accumulates during the texture fetches the sum of the contributing voxels for each sub-region in `sum` register. The dispersion consists basically of two sums, where one is already stored in the `sum` register and the second register (`sq_sum`) is accumulated as sum of squared values of each fetch. After all samples of the sub-region are summed in these registers, the dispersion is computed. We leave out the scaling factor N_S^{-1} to avoid unnecessary computations. Then we compare the minimal dispersion with the currently computed one. If necessary the minimal dispersion value is updated and the corresponding sum is assigned to the `result` register. Finally we normalize the `result` register. The normalization is performed as a multiplication with the N_S^{-1} value, where N_S^{-1} is the number of contributing voxels per sub-region. Finally we write the result to the output. The pseudocode in figure 10 shows the basic steps of rotated mask filtering in graphics hardware.

Initially was our implementation an single-pass approach. The sub-regions are overlapping, i.e., some voxels are contributing to

```

min_disper = HIGH_NUMBER;
result = 0;
for (s = 0; s < 6; s++) {
  sum = sq_sum = 0;
  for all f(i, j, k) in sub_reg[s] {
    sum += f(i, j, k);
    sq_sum += f(i, j, k) * f(i, j, k);
  }
  disper = sq_sum + sum*sum / N_S;
  if (disper < min_disper) {
    min_disper = disper;
    result = sum;
  }
}
return result / N_S;

```

Figure 10: Rotated mask code listing.

more sub-regions. To reduce the number of texture-fetch instructions, we summed the voxel contributions in 6 temporary registers. The compiled HLSL assembly output used 39 temporary registers what strongly influenced the performance. After decomposing the algorithm into 6 passes, the number of temporary registers was reduced to 4 per pass and the performance increased by factor 3.29. The speedup factor compared to a software implementation is 7.26. The original and pre-filtered dataset using the rotating mask filter is shown in figure 11.

5 Hardware-Based Segmentation

For particular segmentation purposes a simple thresholding on a pre-filtered dataset produces acceptable results as compared to more complex methods. Segmentation of the vessel structure of the liver turns out to be a good example. The liver does not have a constant intensity value, and the intensities are close to the intensity of the vessels. The edge-preserving filters smooth the intensities of the liver, but preserve the boundaries of the vessels.

The advantage of GPU-based segmentation is the possibility to interactively adjust the threshold value. The threshold is sent to the fragment program of the segmentation stage. The thresholding is done via one simple conditional that evaluates whether the texture fetch of a source texture is within a given value range. According to the result of this conditional either 0 or 1 is written to the output. After the threshold adjustment is done, we apply the thresholding to the whole dataset and store the segmentation mask in the graphics memory. To keep the segmentation mask as small as possible, we store it in a compact way using for one segmented slice just one bit per pixel of a 32-bit pbuffer. Thus the bit mask of 32 slices can be stored in one pbuffer. The problem is that there is a limited support of pbuffer formats, which are either fixed- or floating-point. Floating-point pbuffers are quite inefficient for storing and restoring the segmentation mask, because it is necessary to distinguish between sign, mantissa and exponent bits. Fixed-point pbuffers are better, however there is not a one-channel pbuffer support available yet. Therefore we store the segmentation mask in four-channel pbuffers, each channel with 8 bits. Current hardware allows to bind just up to 16 textures (data slices) for one rendering pass. To store 32 mask slices we perform two rendering passes. In one pass we render 16 mask slices into the red and green channel. In the next pass we render the next 16 slices into the blue and alpha channel. These two results are blended into a single pbuffer. Such a pbuffer is then either sent to the main memory by the OpenGL function `glReadPixels()`, or stored in the graphics memory in a compact

way. Sending the compressed data to the main memory is much faster than sending 32 slices separately. The restoring (i.e., uncompressing) is easily done on-the-fly. The data slice is multiplied with the corresponding *bit* slice. The segmentation on the graphics hardware compared to a software version is approximately five times faster. Sending compressed data instead of the full dataset is about 20 times faster.

6 Results

We have tested the performance of our algorithms on the newest graphics chip GeForceFX 5900 Ultra [NVIDIA 2003]. The fragment processing unit features 8 texture coordinates and 16 image textures. The textures and puffers support floating-point formats. The maximal length of a fragment program in OpenGL is currently 1024 instructions, which is enough for small operator masks. For filters with larger support we have to split the rendering into multiple passes. The hardware-based filtering is compared to an optimized software version, which was tested on an AMD Athlon XP 2200+ processor with 1.0 GB of RAM (2 × 512 DDR, 133 MHz, CL2) and VIA Apollo KT333 chipset. The size of the test dataset is 512×512×72 of unsigned short, but only 12 bits are set. We perform the filtering on a full precision dataset without any quantization.

In table 1 we summarize the frame rates for filtering a liver dataset. The complex non-linear filters achieve a speedup factor in the range of 1.52 – 7.26. Besides the mentioned non-linear filters, we indicate the performance of the mean and the erosion filter to show the performance improvements compared to previous work from section 3. Such simple filters perform better, because the GPU can execute them more efficiently. The GPU-based segmentation via thresholding achieves a speedup of factor 8.73. Compressing the thresholded dataset does not affect the thresholding performance significantly. The download of the compressed mask to the main memory speeds up the transfer by a factor of 20.

Filter	Hardware [ms]	Software [ms]	Speedup
Mean	1688	19443	11.52
Erosion	1685	25399	15.07
Median	24678	48639	1.97
Bilateral	9668	14706	1.52
Rotated Mask	7989	58003	7.26
Thresholding	40	349	8.73
Thresholding & compression	64	–	–

Table 1: Filtering and threshold-segmentation performance on a liver dataset (512×512×72).

7 Summary and Conclusions

In this paper we presented the implementation of various non-linear filters on state-of-the-art consumer graphics hardware. We used a high-level programming interface for comfortable programming and optimized assembly code. Graphics hardware is now much closer to general purpose hardware, in the way of flexibility.

The overall segmentation pipeline was illustrated; from loading the volumetric dataset into the graphics memory to the compact segmentation mask representation. The compact representation can be used for GPU-based volume rendering in combination with clipping or can be loaded back to main memory in an efficient way. Although the performance of GPU-based complex filters was not

dramatically faster for all filter types comparing to the CPU solutions, there was a certain performance achievement. A nice performance gain was achieved in the segmentation step, where the thresholding, storing and restoring of the segmentation mask was very fast.

We showed that another important part of the visualization pipeline can be performed on the graphics hardware. For particular applications, like for example the liver vessel tree extraction, the whole pipeline can be done on the graphics hardware.

In the paper we pointed out various implementation recommendations to efficiently use the latest graphics hardware. Keeping these issues in mind, it is possible to use the graphics hardware, intended for the gaming industry, for a large spectrum of scientific visualization applications.

The future GPUs will be even more flexible. Hardware-based rendering turned out to be a very effective way for volume visualization [Engel et al. 2001; Guthe et al. 2002]. The problem remains the slow data transfer between CPU and GPU in a hybrid visualization systems. Therefore it is very important to move all possible steps of the visualization pipeline towards the GPU. The performance gain for complex filtering tasks was reasonable though not spectacular. Interactively steered filtering and segmentation on GPU is possible without any transfer between CPU and GPU.

8 Acknowledgments

The work presented in this publication has been funded by the ADAPT project (FFF-804544). ADAPT is supported by *Tiani Medgraph*, Vienna (<http://www.tiani.com>), and the *Forschungsförderungsfonds für die gewerbliche Wirtschaft*, Austria. See <http://www.cg.tuwien.ac.at/research/vis/adapt> for further information on this project. The test dataset is courtesy of Tiani Medgraph. Thanks to Markus Hadwiger, and Jiri Bittner for fruitful discussions and to Michael Knapp for software implementations. Thanks also go to Juan Guardado of the NVIDIA Corporation.

References

- ATI, 2003. ATI web page, <http://www.ati.com/>.
- AURICH, V., AND WEULE, J. 1995. Non-linear Gaussian filters performing edge preserving diffusion. In *Proceedings of 17. DAGM-Symposium*, 538–545.
- DIRECTX, 2003. DirectX web page, <http://www.microsoft.com/directx/>.
- DURAND, F., AND DORSEY, J. 2002. Fast bilateral filtering for the display of high-dynamic-range images. In *Proceedings of ACM SIGGRAPH '02*, 257–266.
- ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, 9–16.
- GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. 2002. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization '02*, 53–59.
- HADWIGER, M., THEUSSL, T., HAUSER, H., AND GRÖLLER, E. 2001. Hardware-accelerated high-quality filtering on PC hardware. In *Workshop on Vision, Modelling, and Visualization VMV '01*, 105–112.
- HADWIGER, M., VIOLA, I., AND HAUSER, H. 2002. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Workshop on Vision, Modelling, and Visualization VMV '02*, 155–162.
- HOPF, M., AND ERTL, T. 1999. Accelerating 3D convolution using graphics hardware. In *Proceedings of IEEE Visualization '99*, 471–474.
- HOPF, M., AND ERTL, T. 2000. Accelerating morphological analysis with graphics hardware. In *Workshop on Vision, Modelling, and Visualization VMV '00*, 337–345.

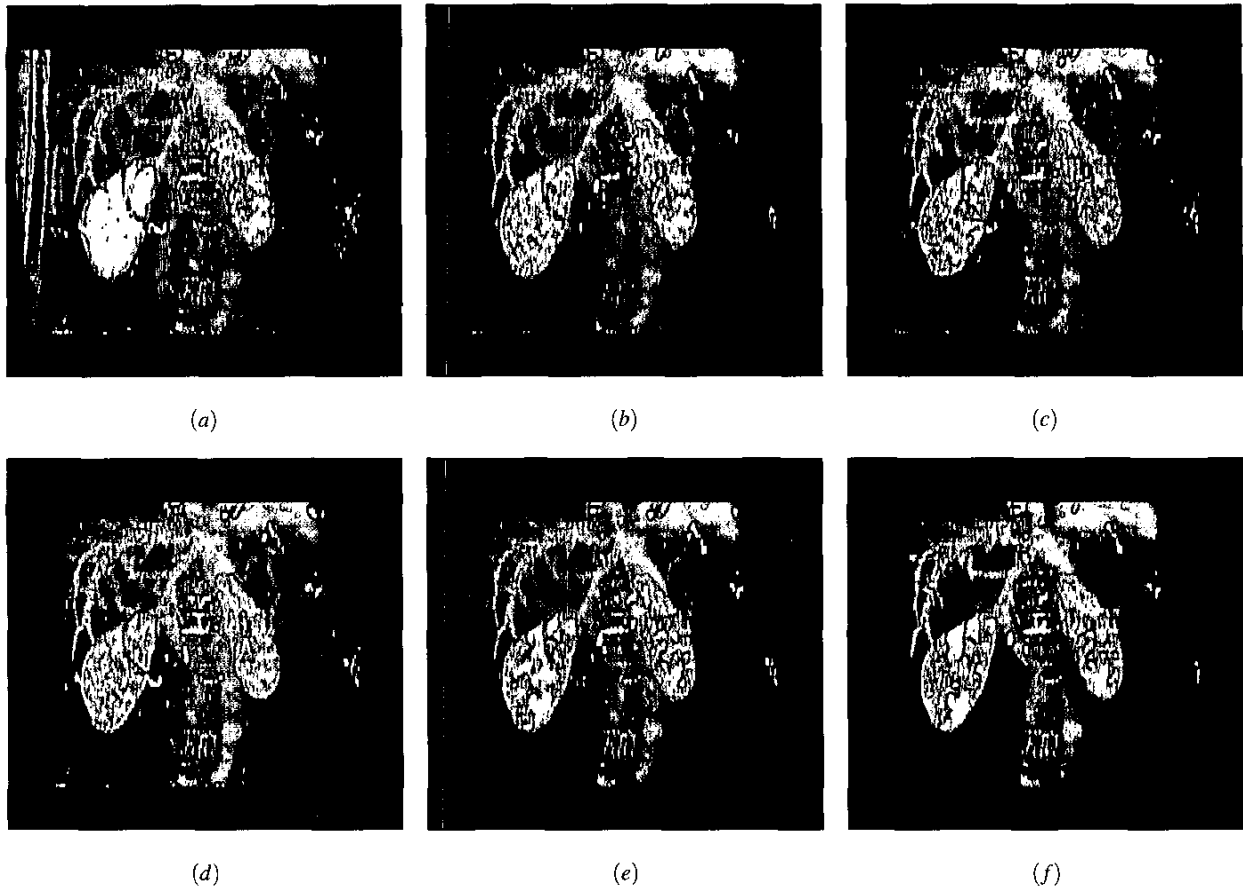


Figure 11: Visualization of liver vessel structure. The pre-filtered dataset was segmented via thresholding to generate a mask for vessels within the liver. This mask was applied to original dataset. We show the results of segmentation without pre-filtering (a), with pre-filtering using median (b), rotating mask (c) and bilateral (d) (e) (f) filters. The mask generated by pre-filtering using the bilateral filter is shown after 1, 3, and 5 filtering iterations.

HUANG, T., YANG, G., AND TANG, G. 1979. A fast two-dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 27, 13–18.

JAMES, G. 2001. Operations for hardware-accelerated procedural texture animation. In *Game Programming Gems 2*, Charles River Media, M. DeLoura, Ed., 497–509.

MARK, W., GLANVILLE, S., AKELEY, K., AND KILGARD, M. 2003. Cg: A system for programming graphics hardware in a C-like language. In *Proceedings of ACM SIGGRAPH'03*, 896–907.

NAGAO, M., AND MATSUYAMA, T. 1979. Edge preserving smoothing. *Computer Graphics and Image Processing* 9, 394–407.

NVIDIA, 2003. NVIDIA web page, <http://www.nvidia.com/>.

OPENGL, 2003. OpenGL web page, <http://www.opengl.org/>.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 2002. *Numerical Recipes in C*. Cambridge University Press.

PROUDFOOT, K., MARK, W., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH'01*, 159–170.

SONKA, M., HLAVAC, V., AND BOYLE, R. 1995. *Image Processing, Analysis and Machine Vision*. PWS Publishing.

TASDIZEN, T., WHITAKER, R., BURCHARD, P., AND OSHER, S. 2002. Geometric surface smoothing via anisotropic diffusion of normals. In *Proceedings of IEEE Visualization '02*, 125–132.

TOMASI, C., AND MANDUCHI, R. 1998. Bilateral filtering for gray and color images. In *Proceedings of IEEE International Conference on Computer Vision*, 839–846.

UPSTILL, S. 1990. *The RenderMan Companion*. Addison-Wesley.

WEISKOPF, D., ENGEL, K., AND ERTL, T. 2002. Volume clipping via per-fragment operations in texture-based volume visualization. In *Proceedings of IEEE Visualization '02*, 93–100.