Quality Issues of Hardware-Accelerated High-Quality Filtering on PC Graphics Hardware

Markus Hadwiger* Helwig Hauser* Torsten Möller†

*VRVis Research Center
Vienna, Austria
http://www.VRVis.at/vis/
{Hadwiger, Hauser}@VRVis.at

†Graphics, Usability, and Visualization Lab
Simon Fraser University, Canada
http://gruvi.cs.sfu.ca/
torsten@cs.sfu.ca

ABSTRACT

This paper summarizes several quality issues of an approach for high-quality filtering with arbitrary filter kernels on PC graphics hardware that has been presented previously. Since this method uses multiple rendering passes, it is prone to precision and range problems related to the limited precision and range of intermediate computations and the color buffer. This is especially crucial on consumer-level 3D graphics hardware, where usually only eight bits are stored per color component. We estimate the accumulated error of several error sources, such as filter kernel quantization and discretization, precision of intermediate computations, and precision and range of intermediate results stored in the color buffer. We also describe two approaches for improving precision at the expense of a higher number of rendering passes. The first approach preserves higher internal precision over multiple passes that are forced to store intermediate results in the less-precise color buffer. The second approach employs hierarchical summation for attaining higher overall precision by using the available number of bits in a hierarchical fashion. Additionally, we consider issues such as the order of rendering passes that is crucial for avoiding potential range problems, and a variant of hardware-accelerated high-quality filtering that is able to reduce the number of passes by four for filtering single-valued data, thus improving both performance and precision.

Keywords

texture filtering, hardware convolution, precision issues, graphics hardware

1 Introduction

Many recent real-time rendering algorithms are able to achieve high quality results by using many rendering passes. These methods accumulate intermediate results in the frame buffer in order to generate the final image.

One common problem shared by all of these approaches is the limited range and precision of hardware frame buffers. Usually, graphics hardware offers only eight bits per color component stored in the frame buffer, severely limiting the potential of color buffers for storing intermediate results of general computations. Additionally, the common $[0, 1]$ range even further complicates storing intermediate results in hardware color buffers, since not even a sign bit is available and large values cannot be stored directly. A common method for tackling the range problem is to employ scale and bias operations to fit the needed range into the constraints of the hardware. However, this further exacerbates the problems related to limited precision.

Recently, interest in higher precision and range for storing intermediate results of computations in graphics hardware has increased noticeably, especially since the introduction of real-time shading languages [12, 13]. Apart from simply extending the precision and range of frame buffers themselves, alternative approaches such as F-buffers [6] have also been suggested. Only with the most recent hardware architectures (ATI Radeon 9700 and NVIDIA NV30), the process of moving towards floating-point color computations has begun. However, such hardware is far from being widely available, or not yet available at all.

Although the precision in color buffers of most available hardware is severely limited, recent hardware employs higher precision and range for internal computations before finally storing the unsigned eight-bit result in the frame buffer. The NVIDIA GeForce 3 and 4 officially use eight bits plus one sign bit internally, although our experimental results suggest that their internal precision is actually higher. Intermediate computations are performed in a range of $[-1, 1]$. The ATI Radeon 8500 internally uses twelve bits for the fractional part, plus four bits for integer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Journal of WSCG, Vol.11, No.1., ISSN 1213-6972
WSCG'2003, February 3-7, 2003, Plzen, Czech Republic.

part and sign, thus achieving an extended range of $[-8, 8]$.

Extended internal precision and range can be exploited for higher quality rendering by choosing an order of rendering passes that minimizes the impact of the limited external precision as much as possible. Additionally, higher internal precision can be preserved between passes by splitting up the intermediate results, only generating the final result in a final combination pass. The drawback of this, however, is a higher number of rendering passes.

We are especially interested in estimating the error incurred by limited precision and range in an approach for using arbitrary filter kernels for high-quality filtering on graphics hardware [3, 4, 5]. We describe the sources of numerical error, and present results of a numerical simulation of the errors incurred by filter kernel quantization and discretization, according to different sampling resolutions and filter types. Kernel sampling parameters are a crucial issue in the approach we are interested in, since it samples and stores the actual continuous filter kernel into multiple texture maps. The algorithm treats the kernel as though it were continuous, although it is stored in a discrete representation, and reconstruction is used at run-time in order to retrieve weights from the "original" continuous filter function on-the-fly.

2 Estimating the error of hardware-accelerated filtering

This section analyzes hardware-accelerated high-quality filtering with regard to the different sources of error (especially numerical error).

The filter convolution sum (equation 1) is evaluated in a multi-pass rendering algorithm, which is prone to artifacts due to precision and range limitations of the hardware color buffer and internal computations.

$$g(x) = f[x] * h(x) = \sum_{i=\lfloor x \rfloor - \lceil m \rceil + 1}^{\lfloor x \rfloor + \lceil m \rceil} f[i]h(x-i) \qquad (1)$$

This equation describes a convolution of the discrete input samples $f[x]$ with a continuous reconstruction filter $h(x)$, yielding the reconstructed output function $g(x)$. The (finite) half-width of the filter kernel is denoted by m.

2.1 Error sources

Related to the evaluation of the filter convolution sum (equation 1), we distinguish the following sources of error.

First, error related to the filter kernel used and its representation in hardware:

- Since the filter kernel is sampled and stored in several texture maps, it has to be quantized to the bit-resolution of these textures. That is, the weights in the filter kernel are represented by b bits (where usually $b = 8$).
- Naturally, storing the filter kernel in texture maps also requires discretization, i.e., sampling the kernel at discrete locations. In the approach we are analyzing, the maximum texture resolution of the hardware can be used for each extent of unit size in the filter kernel. Thus, the width of the filter kernels that can be employed is not restricted by hardware texture size limits. However, in practice the sampling frequency used for the filter kernel itself is limited by the texture memory consumed.
- When retrieving weights from the filter kernel, the corresponding texture has to be queried, also employing reconstruction. For this, either point-sampling or the hardware-native linear interpolation is employed, which introduces further error in the filter weights that are actually used.
- The filter kernel also incurs an "inherent" non-numeric error, depending on its type and width. That is, a given kernel introduces a certain reconstruction error, even if we would be able to represent it as a continuous function, instead of a collection of discrete values. Möller et al. [8, 9, 10] present a framework for estimating filter kernel-native errors, as well as error bounds for several interesting types of filter kernels.

Second, error is introduced by the evaluation of the convolution sum in hardware using fixed-point arithmetic (usually mapping 1.0 to $2^b - 1$ instead of 2^b, and 0.0 to 0, in order to make maximum use of the available number of bits):

- Precision is lost in the multiplication of input samples with filter weights, since it introduces an error of its own (in contrast to addition, see below). The overall error therefore greatly depends on the precision with which multiplication is performed, and the number of multiplications needed for generating the final result, which, in our case, is equal to the number of rendering passes.
- Addition only propagates the error of its input operands if we assume that no clamping occurs due to range issues (see below). The error analysis in this section assumes that addition does not introduce new error by itself.
- If internal computations are carried out at a higher precision than the one available for storing intermediate results in the color buffer, further error that depends on the number of rendering passes required is introduced.

Third, additional – and usually quite severe – error may be introduced when the color buffer range is exceeded for intermediate results, leading to undesired clamping (i.e., values being forced into the $[0, 1]$ range by saturation). Therefore, it is crucial to choose an evaluation order of rendering

passes that avoids leaving the available range for intermediate results. In practice, this means that intermediate results must never be below 0.0, or above 1.0. See section 4 for a description of how clamping artifacts can be avoided.

2.2 Fixed-point representation

Computations on color values carried out by graphics hardware are usually done in a fixed-point representation. However, in order to use the entire dynamic range available for a given number of bits to represent the floating point range of $[0.0, 1.0]$ and still have an exact representation of 1.0, the interval $[0.0, 1.0]$ is usually mapped to $[0, 2^b - 1]$ instead of $[0, 2^b]$, see, e.g., [1, 11].

That is, as opposed to the usual fixed-point approach of mapping 1.0 to a power of two, it is mapped to the highest number representable with a certain number of bits. This maximizes utilization of the available number of bits, but somewhat complicates arithmetic operations in hardware, see section 2.4.

Mapping a floating-point color value x to a fixed-point color value $\bar{x}$ is thus done like this if truncation is used:

$$\bar{x}_{(truncated)} = \lfloor x * (2^b - 1) \rfloor \quad (2)$$

and like this if rounding is used:

$$\bar{x}_{(rounded)} = \lfloor x * (2^b - 1) + 0.5 \rfloor \quad (3)$$

Note that this special mapping leads to a slightly different quantization error than the one usually used (i.e., one lsb), see below.

2.3 Filter kernel error (ϵ_h)

We denote the error incurred by the representation of the filter kernel in texture maps by ϵ_h, and distinguish three sources of error that contribute to it.

First, due to the necessary quantization of input values to b bits, the following maximum error is introduced. If the high-precision input value is simply truncated to fit into the available number of bits, we get a quantization error $\epsilon_{q(truncated)} = \frac{1}{2^b - 1}$. Note that this is not exactly the usual one lsb (least significant bit, i.e., 2^{-b}), due to the special mapping described in the previous section. If rounding is employed, we get $\epsilon_{q(rounded)} = \frac{0.5}{2^b - 1}$. Further, error introduced by the limited sampling resolution (the discretization) and the reconstruction used for the filter kernel (either point-sampling via GL_NEAREST, or linear interpolation via GL_LINEAR) are two additional sources of numerical error that contribute to ϵ_h and must be considered.

Instead of actually considering these three sources of error separately, we calculate an estimation of the overall numerical error ϵ_h using a numerical simulation of the conditions that will be used in the hardware-accelerated algorithm. We sample the filter kernel at a specified resolution and quantize the sampled values to the number of bits that will be used in the actual texture maps. Using the reconstruction method that will be employed by the hardware to retrieve filter weights from the filter textures (either point-sampling or linear interpolation), we derive approximately the same values that will be generated by the hardware and compare them with the corresponding reference values from the analytically represented filter kernel.

In this way, we are able to estimate an error bound subsuming all three sources mentioned above for a single filter weight. In order to estimate the error introduced by the filter kernel into the entire evaluation of the filter convolution sum, however, we need to account for the error of all filter weights. For instance, filtering with a cubic kernel in one dimension uses four different weights, retrieved from the kernel at locations spaced one unit apart.

For a given resampling location i, the error incurred by all weights can be calculated as $\epsilon_h^i = \sum_j |\epsilon_{h,j}|$, with $\epsilon_{h,j}$ denoting the actual filter weights from the actual filter kernel under consideration that correspond to the given resampling location i. In order to estimate the error for all possible resampling locations, we calculate such ϵ_h^i at a high number of resampling locations and take the maximum, thus:

$$\epsilon_h = \max_i \, (\epsilon_h^i = \sum_j |\epsilon_{h,j}|) \quad (4)$$

Table 1 shows values of ϵ_h for certain scenarios, where ϵ_h subsumes the numerical errors due to quantization (ϵ_q), discretization, and filter kernel reconstruction in hardware. The corresponding filter functions are depicted in figure 1.

2.4 Computation error (ϵ_m)

In this section, we consider the error incurred by the evaluation of the convolution sum itself. This evaluation employs only two different kinds of arithmetic operations, namely addition and multiplication. We will see that the entire error due to the computation itself is introduced by the multiplication, ϵ_m.

Fixed-point arithmetic addition in hardware is usually simply done as $\bar{x} + \bar{y}$, since:

$$\bar{x} \oplus \bar{y} = x(2^b - 1) + y(2^b - 1) = (x + y)(2^b - 1) \quad (5)$$

The addition of two b bit values yields at most $b + 1$ bits in the result and there is no new error being introduced (assuming the result still fits into the available number of bits, thus avoiding undesired clamping).

Fixed-point arithmetic multiplication in hardware is usually done as $\lfloor (\bar{x}\bar{y})/(2^b - 1) + 0.5 \rfloor$, since:

$$\bar{x} \otimes \bar{y} = \frac{x(2^b - 1) * y(2^b - 1)}{2^b - 1} = (x * y)(2^b - 1) \quad (6)$$

The multiplication of two b bit values yields at most $2b$ bits in the result. Due to the division by $(2^b - 1)$ needed to normalize the result, a new error is introduced by the multiplication operation itself, even if the input values were exact.

The error that is introduced by the multiplication can be bounded by (for either truncated, or rounded results, re-

filter kernel type	tile resolution [1D samples/tile]	2D		3D	
		$255 * \epsilon_{h(box)}$	$255 * \epsilon_{h(lin)}$	$255 * \epsilon_{h(box)}$	$255 * \epsilon_{h(lin)}$
Cubic B-spline ($B = 1.0, C = 0.0$)	16	11.2351	5.5803	12.3400	7.2346
	32	6.1619	3.9219	7.1811	5.4758
	64	3.8838	3.3165	5.0478	7.0751
	128	2.8293	2.5841	4.4375	4.1687
	256	2.3145	2.2676	*n/a*	*n/a*
	512	2.0867	2.0867	*n/a*	*n/a*
Catmull-Rom ($B = 0.0, C = 0.5$)	16	25.3076	15.0835	32.6554	19.1470
	32	14.4100	10.8905	17.8089	14.3186
	64	8.7091	8.3325	11.2476	11.9767
	128	6.0308	7.1140	8.1039	10.5024
	256	4.9057	6.7169	*n/a*	*n/a*
	512	4.3535	6.3613	*n/a*	*n/a*
Blackman sinc (window width 4)	16	26.3035	15.4278	34.0124	20.0338
	32	14.3229	10.7960	17.9082	14.7748
	64	8.8159	8.3062	11.4056	12.0028
	128	6.0954	7.2064	8.1723	10.6158
	256	4.9426	6.6606	*n/a*	*n/a*
	512	4.3151	6.3229	*n/a*	*n/a*

Table 1: Filter kernel error bounds (ϵ_h) for different scenarios. All kernel weights have been quantized to eight bits, and the error bounds are absolute errors in a $[0, 1]$ domain, multiplied by 255. $\epsilon_{h(box)}$ uses a box filter for kernel reconstruction, $\epsilon_{h(lin)}$ linear interpolation. In the 2D case, the error has been estimated through 1024^2 resampling locations, whereas in the 3D case 256^3 resampling locations have been used (corresponding to the number of different values for i in equation 4). For comparison, just taking quantization to eight bits into account, a maximum absolute error of 0.5 would be incurred per filter weight, yielding error bounds of $8 = 0.5 * 16$ in two dimensions, and $32 = 0.5 * 64$ in three dimensions, respectively. Thus, the numbers in this table show that the error incurred by filter kernel representation in reality is much less than the overly conservative estimate of adding up the upper quantization error bounds of individual filter weights, e.g., 2.82 instead of 8 (bi-cubic B-spline, sampled with 128 samples per dimension and tile). In our experience, absolute errors of about 3-5 achieve results of sufficient optical quality. Usually, we are using 64 samples for a cubic B-spline, and 128 samples for Catmull-Rom splines and Blackman-windowed sincs, and simple nearest-neighbor interpolation for kernel reconstruction. *n/a* entries have not been measured, since 3D kernels of the corresponding sizes are infeasible, and the numerical simulation consumes a considerable amount of time.

spectively):

$$\epsilon_{m(truncated)} = \frac{2^b - 2}{(2^b - 1)^2} < \frac{1}{(2^b - 1)} \quad (7)$$

$$\epsilon_{m(rounded)} = 0.5\frac{2^b - 2}{(2^b - 1)^2} < \frac{0.5}{(2^b - 1)} \quad (8)$$

For example, two cases interesting to us are:

$$\epsilon_{m(rounded,8)} = 0.0019608 \quad (9)$$

$$\epsilon_{m(rounded,12)} = 0.0001221 \quad (10)$$

for eight and twelve bits of precision, respectively.

Note that, although a division by a non-power-of-two value is theoretically necessary, the correct result – even including rounding – can for instance be generated without a division as follows (for $b = 8$, see [1]):

```
i = x*y + 128;
r = ( i + ( i >> 8 ) ) >> 8;
```

2.5 Accumulated error

Now that we have bounded the individual errors involved, we can determine a worst-case bound for the overall evaluation of the filter convolution sum.

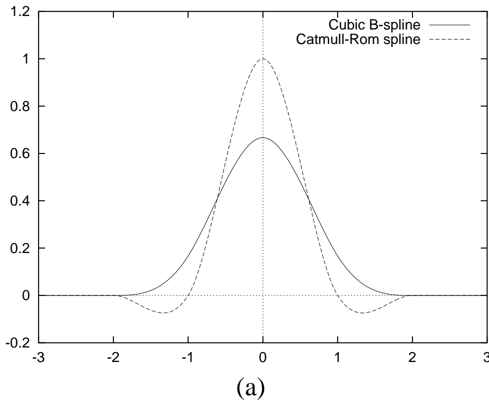
In the following, we assume that the input sample values are exact, the errors of the weights retrieved from the filter kernel are bounded by ϵ_h, and the error introduced by each multiplication in the convolution sum is bounded by ϵ_m.

The numerical errors introduced during evaluation of the filter convolution sum can be estimated as follows (denoting the number of rendering passes by N, and using $-\epsilon_m < \epsilon'_m < \epsilon_m$ and $-\epsilon_h < \epsilon'_h < \epsilon_h$ to denote actual errors corresponding to a given rendering pass as opposed to error bounds):

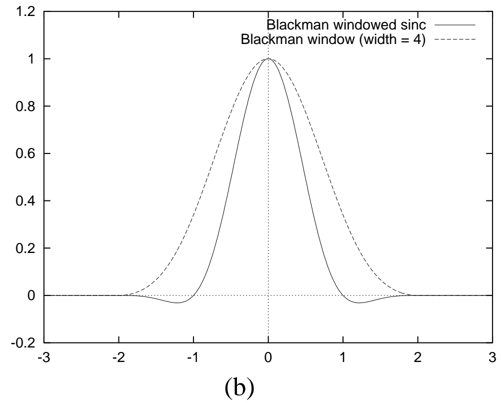
$$\sum_N f \otimes h_{texture} = \sum_N [f * (h + \epsilon'_h) + \epsilon'_m] \quad (11)$$

$$= \sum_N f * h + \sum_N f * \epsilon'_h + \sum_N \epsilon'_m \quad (12)$$

We now denote the overall error due to the numerical filter



(a)



(b)

Figure 1: Filter kernels we have used: (a) Cubic B-spline and Catmull-Rom spline; (b) Blackman-windowed sinc depicting also the window itself.

kernel representation, and the error introduced by the fixed-point multiplication by E_2 and E_3, respectively:

$$\sum_N f \otimes h_{texture} = \sum_N f * h + E_2 + E_3 \qquad (13)$$

That is, these errors directly depend on the number of rendering passes needed for the evaluation of the convolution sum (N), and we get:

$$E_3 < N\epsilon_m \qquad (14)$$

Assuming the function itself (f) is bounded by 1.0, we get:

$$E_2 < N\epsilon_h \qquad (15)$$

Further introducing the error due to the filter itself (even if represented without numerical error, i.e., comparing the actual filter h to the ideal *sinc* filter) as E_1, we get:

$$\sum_N f \otimes h_{texture} = \sum_{-\infty}^{+\infty} f * \mathit{sinc} + E_1 + E_2 + E_3 \qquad (16)$$

The error E_1 depends on the filter itself (i.e., its deviation from the behavior of a windowed sinc), and can be calculated using a Taylor series expansion of the convolution sum [9].

The most crucial restriction is the value of E_3, since it is entirely determined by the hardware. The other two errors can be chosen up to a certain extent. If possible, we try to do this in the following way:

- Choose the filter kernel such that $E_1 < E_3$, if possible.
- Choose the kernel texture resolution such that $E_2 < E_3$.

In practice, though, subjective visual judgment is the single most important criterion for selecting parameters, since eight to twelve bits of precision do not leave much space for considerations with conservative error bounds. In reality, visual results are still good where error estimation would suggest that too little precision has been used, and choosing the filter kernel sampling rate becomes most important.

3 Increasing precision for intermediate results

This section summarizes two approaches for gaining higher precision for the computation of intermediate results. The first approach strives to preserve higher internal precision supported by the graphics hardware across rendering passes that have to store intermediate results in the 8-bit frame buffer. It does not mandate any knowledge about the numerical subrange that will actually be used by a given set of rendering passes. The second approach assumes no higher internal precision, but performs summation in a hierarchical way that is also able to achieve higher precision results. However, it requires to know the numerical subrange that is actually used by a given set of rendering passes. Another approach that can be used to improve precision on lower-precision hardware has also been suggested recently [14].

3.1 Preserving internal precision across passes

Provided that internal computations are done by the hardware in higher precision than the external precision of the color buffer, a multi-pass approach can be used in order to preserve this precision across passes to a certain extent, even if temporaries need to be stored in the color buffer. In this context, we are exclusively dealing with addition. Multiplication is always performed with internal precision anyway, so we care about adding up the individual terms with higher precision than the frame buffer supports directly.

The basic idea is to perform the entire computation (i.e., evaluation of the convolution sum) twice. We denote the number of bits for internal computations by b and split it up into two bit-adjacent parts b_i and b_j, so that $b = b_i + b_j$. We also specify that the b_i bits contain the msb (most significant bit), and the b_j bits contain the lsb (least significant bit). If we denote the number of external bits (i.e., the precision of the frame buffer) by m, we choose $b_i = m$, and require that $b_j \leq m$. Since the resources of the frame

buffer are more scarce than internal pixel paths, and calculating results in less precision than available for storing them makes no sense, $b \geq m$ of course also holds.

Now, we proceed by calculating two intermediate results and combining them afterwards. First, the desired computation is done for the part with b_i bits, storing the result in an off-screen buffer (either by rendering directly into a texture, or rendering into the framebuffer and copying it into a texture afterwards). Next, the same computation is done again, but this time for the part with b_j bits, yielding a certain number of carry bits, which we will be denoting by b_k. Finally, a single combination pass combines the two separate intermediate buffers, correctly taking the carry bits into account, and producing the final result. Thus, if the computation usually needs N rendering passes, we now need $2N+1$ passes if we want to preserve the internal precision across passes. Note that if rendering directly to a texture is not supported, we only need to copy the frame buffer into a texture twice, independent of N. The reason for this is that we can do all computations in the frame buffer, except for the two input textures to the final combination pass, which need to be acquired.

In practice, we are most of all restricted by the number of carry bits that are generated and need to be stored. Over N passes, we create $b_k = \lfloor \log_2 N \rfloor$ carry bits, which have to fit into m bits together with the b_j base bits. Thus, $m \geq b_j + \lfloor \log_2 N \rfloor$ and if we want to preserve all internal bits, we are able to do so over at most $N = 2^{m-b_j+1} - 1$ passes.

We now give two examples with actual hardware-dependent numbers. First, on the GeForce 3 and 4, we assume that $b = 9$ and $m = 8$. Thus, $b_i = 8$, $b_j = 1$, and we can preserve all internal bits over at most $N = 255$ rendering passes, which would yield $b_k = 7$. However, we deem the associated performance impact for just one additional bit of precision too high. Second, on the Radeon 8500, we assume that $b = 12$ and $m = 8$. Thus, $b_i = 8$, $b_j = 4$, and we can preserve all internal bits over at most $N = 31$ rendering passes, which would yield $b_k = 4$. Further, for the interesting case of $N = 64$ (tri-cubic reconstruction of volume data), we can still preserve $m' = 10$ bits of internal precision ($m' \leq m$), which we still deem well worth the additional effort.

Another very implementation-dependent issue of the method outlined above, is how the internal results are actually split up into the two parts b_i and b_j, requiring bit shifting and masking, and how the combination pass is actually implemented, requiring bit shifting. We achieve bit shifting by possibly multiple multiplications with a scale factor, exploiting fixed scale and bias functionality of the graphics hardware. Factors less than one can also be achieved by multiplying with an arbitrary constant color, but multiplication with factors larger than one are not possible in this way, and require explicit support for scaling. The supported scale factors are hardware-dependent. Bit masking is usually not supported explicitly, and we achieve this by first shifting left, and then shifting right again, exploiting the automatic clamping to get rid of unwanted most significant bits. An alternative way would be to mask out the desired bits, and subtracting the undesired bits from the original value.

In the combination pass, the b_k carry bits produced in the computation corresponding to the b_j least significant bits have to be added to the part containing the result of the b_i most significant bits. For this, bit shifting is also required. That is, the carry bits have to be extracted from the intermediate result, and shifted into the proper position for addition.

3.2 Hierarchical summation

An approach for increasing the precision of intermediate computations, even given a limited internal precision, is to add results in a hierarchical manner.

Let b denote the number of bits available throughout the entire computation (i.e., both internally and externally). If we know that the filter weights used in a set of passes never exceed a certain threshold, we can pre-multiply all of these values in order to maximize usage of the available range, gaining "additional" bits of precision that would have been lost otherwise. The result of each of such a set of passes is accumulated in a corresponding off-screen rendering buffer. Renormalization to the actual range is then performed when compositing these intermediate results to generate the final image.

This approach is especially simple to realize in the case of 1:1 filtering (e.g., image processing), where there is only a very small number of filter weights. For example, the 16 passes associated with a 4x4 (2D) averaging filter with equal weights can be split up in batches of four passes, pre-multiplying the filter weights by 4 ("gaining" two bits of precision). There are four such batches, correspondingly generating four intermediate results. These have to be scaled by 0.25 and composited in a final combination pass.

Note that rendering to separate frame buffers for intermediate results can be achieved efficiently on many current graphics hardware architectures by using the `WGL_ARB_render_texture`, `WGL_ARB_pbuffer`, and `WGL_ARB_pixel_format` extensions, avoiding the slow `glCopyTexSubImage2D()` call.

3.3 Logarithm addition

We would like to briefly mention an idea brought up by Michael McCool [7] for gaining more range in multiplications. If we store the logarithm of input values to multiplications, we can add these values instead of multiplying them, and use a texture as lookup table for exponentiation in order to convert back to the actual output values when we need them.

Jim Blinn [2] also describes why floating point numbers are essentially a logarithmic representation.

4 Rendering pass order

The order of rendering passes is crucial to avoiding unintentional clamping of intermediate results against the $[0, 1]$ range of the frame buffer.

We have therefore implemented a numerical simulator for the range behavior of a certain pass order. That is, we assume the worst-case input data of 1.0 everywhere, and accumulate the values contained in filter tiles for each sample location separately. The maximum and minimum values can be observed during accumulation. Using this facility makes it possible to ascertain beforehand whether a certain pass order is able to avoid clamping errors or not. If the maximum and minimum values never go above 1.0 or 0.0, respectively, during accumulation, the pass order can be used at run-time.

Additionally, if it is not possible to find a pass order that avoids clamping, we split up filter tiles into two sub-tiles, one of them containing the larger values of the tile, the other one containing the smaller values. These two separate tiles can be inserted into the pass order at arbitrary (and non-adjacent) locations, which allows to avoid clamping, even if this would not have been possible otherwise. Naturally, this increases the number of rendering passes that are required.

5 Rendering pass bias

When we want to directly reconstruct gradients, instead of original function values, bias values have to be added in each pass. For example, the vector component values in normalized gradients are between -1 and 1. However, in hardware rendering, this is mapped to a $[0.0, 1.0]$ range by scaling by 0.5, and adding a bias of 0.5.

We add individual bias values in each rendering pass, which altogether sum up to the required overall bias of 0.5. The reason for this is once again to avoid clamping errors between passes. A single addition of a single 0.5 bias allows no fine-control over when (during passes) negative values can be avoided by adding a small bias, but simultaneously not adding so much as to go over 1.0.

6 The dot4 algorithm

This section describes a method that is able to fold four theoretical rendering passes into a single actual pass by interleaving a monochrome source texture four times with itself in the RGBA components of a four-component texture, matching it with analogously pre-interleaved filter tile textures, and exploiting hardware dot products for performing the corresponding multiplications and additions.

In general, the major problem with evaluating the filter convolution sum in hardware is getting at all the necessary input data, and the associated bandwidth requirements. Remember that each rendering pass corresponds to a specific offset of the input texture, which is point-sampled in order to use the correct values in the multiplication with the corresponding filter weight. But usually we only have access to one such value per pixel in a single rendering pass (assuming 2x multi-textured texture mapping; the second texture is required for the filter tile texture). However, if we restrict ourselves to monochrome input textures, we can exploit the fact that a single texel retrieved by the hardware actually consists of four values, namely the R, G, B, and A components of the texel.

We can leverage this fact by taking a monochrome input texture, and turning it into an RGBA texture by pre-applying four selected input offsets. We store the first pre-offset texture into the R component, the second into the G component, the third into the B component, and the fourth into the A component.

If we sample a single texel of this texture at run-time, we get simultaneous access to the input data usually associated with four different rendering passes. The respective part of the convolution sum is evaluated by exploiting the capability of recent graphics cards to perform per-pixel dot products on color vectors. Thus, we call this algorithm the *dot4 algorithm*, since it employs a four-vector dot product. Note that the four-vector dot product might have to be substituted by a three-vector dot product (RGB only), and adding separately multiplied alpha values. This is the case on the GeForce 3 and 4, for instance, where the RGB dot product is calculated in the RGB portion of a general combiner, the A product in the ALPHA portion, and, unfortunately, an additional combiner stage is needed for adding these two partial results.

Furthermore, in order to match the interleaved source texture with the filter kernel, an analogous interleaving scheme must be applied to the filter tile textures. This is not as trivial as it might seem at first glance, and we will have a more detailed look at this problem below.

The two major drawbacks of the dot4 algorithm are that it is restricted to monochrome input data, and that the pre-interleaved texture consumes four times the memory of the original monochrome texture. However, a colored texture of the same size would consume at least 0.75 times the amount of memory (RGB instead of RGBA).

However, it still has much to offer, in that it is able to reduce not only the number of passes by a factor of four, but also the texture bandwidth on the graphics card (compared to a colored texture), thus practically quadrupling the performance. The texture bandwidth reduction is an especially big advantage compared to solely exploiting multi-texturing architectures supporting more than two simultaneous textures, since these can only reduce the bandwidth to the frame buffer, but not to the texture memory, which is much more crucial in practice. Still, the fewer passes the better, and in practice we of course use a combination of the highest number of simultaneous textures possible, together with the dot4 approach. Apart from performance reasons, this is also crucial to maximize utilization of the available internal precision.

As mentioned above, as soon as an interleaved source texture is used, the filter tile textures also have to be interleaved in order to match the same values as in the basic

(non-interleaved) algorithm. The interleaving of filter tiles is closely related to the choice of offsets that have been applied to generate the interleaved input texture. Since only a single interleaved input texture is used, it must be possible to generate all other offsets (corresponding to one pass each) by simply adding another offset to the interleaved input texture on-the-fly. That is, each offset applied during rendering automatically generates four actual offsets, and all possible offsets must be generated exactly once.

7 Conclusions and future work

In this paper, we have described the different sources of numerical error that are relevant in hardware-accelerated high-quality filtering with arbitrary filter kernels sampled into multiple texture maps. The numerical simulations that we have presented help to determine an order of rendering passes that avoids unintentional clamping of intermediate results, and allow to estimate the interdependence of filter kernel sampling resolution, filter kernel type, and quantization. As was to be expected from the frequencies contained in those filter kernels, the cubic B-spline causes the least numerical problems, followed by the Catmull-Rom spline and a Blackman-windowed sinc, respectively. We have also estimated the overall error incurred during evaluation of the filter convolution sum in hardware, taking the above-mentioned error sources into account.

Future work in this context would include additional filter kernels, and comparisons of inherent filter kernel errors and their relation to errors incurred in the algorithm itself.

Please see http://www.VRVis.at/vis/research/hq-hw-reco/ for more information on this ongoing research project.

8 Acknowledgments

Parts of this work have been carried out as part of the basic research on visualization at the VRVis Research Center in Vienna, Austria (http://www.VRVis.at/vis/), which is funded in part by an Austrian research program called Kplus.

The first author would like to express very cordial thanks to the third author for a very nice research stay at Simon Fraser University, Burnaby, B.C., during September 2001. We would like to especially thank Thomas Theußl for much-appreciated help with preparing this paper and many stimulating discussions. We would also like to thank Wolfgang Heidrich for very interesting discussions, and Larry Seiler of ATI for suggesting that the higher internal precision of the Radeon 8500 can be preserved across passes.

References

[1] J. F. Blinn. Three wrongs make a right. *IEEE Computer Graphics and Applications*, 15(6), 1995.

[2] J. F. Blinn. Floating-point tricks. *IEEE Computer Graphics and Applications*, 17(4), 1997.

[3] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Hardware-accelerated high-quality filtering of solid textures. In *Conference Abstracts and Applications, SIGGRAPH 2001*, page 194, 2001.

[4] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Hardware-accelerated high-quality filtering on PC hardware. In *Proceedings of Vision, Modeling, and Visualization Workshop 2001*, 2001.

[5] M. Hadwiger, I. Viola, T. Theußl, and H. Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Proceedings of Vision, Modeling, and Visualization Workshop 2002*, 2002.

[6] W. R. Mark and K. Proudfoot. The f-buffer: A rasterization-order fifo buffer for multi-pass rendering. In *Proceedings of Eurographics/SIGGRAPH Graphics Hardware Workshop 2001*, pages 57–64, 2001.

[7] M. McCool. Homomorphic factorization for brdfs. In *Proceedings of SIGGRAPH 2001*, pages 171–178, 2001.

[8] T. Möller, R. Machiraju, K. Müller, and R. Yagel. Classification and local error estimation of interpolation and derivative filters for volume rendering. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 71–78, 1996.

[9] T. Möller, R. Machiraju, K. Müller, and R. Yagel. Evaluation and Design of Filters Using a Taylor Series Expansion. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):184–199, 1997.

[10] T. Möller, K. Müller, Y. Kurzion, Raghu Machiraju, and Roni Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 143–151, 1998.

[11] A. W. Paeth. Proper treatment of pixels as integers. *Graphics Gems I*, pages 249–256, 1990.

[12] M. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proc. of SIGGRAPH 2000*, pages 425–432, 2000.

[13] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proc. of SIGGRAPH 2001*, pages 159–170, 2001.

[14] R. Strzodka. Virtual 16 bit precise operations on rgba8 textures. In *Proceedings of Vision, Modeling, and Visualization Workshop 2002*, 2002.