Virtual 16 Bit Precise Operations on RGBA8 Textures

R. Strzodka

Numerical Analysis and Scientific Computing

University of Duisburg, Germany

Email: strzodka@math.uni-duisburg.de

Abstract

There is a growing demand for high precision texture formats fed by the increasing number of textures per pixel and multi-pass algorithms in dynamic texturing and visualization. Therefore support for wider data formats in graphics hardware is evolving. The existing functionality of current graphics cards, however, can already be used to provide higher precision textures. This paper shows how to emulate a 16 bit precise signed format by use of RGBA8 textures and existing shader and register operations. Thereby a 16 bit number is stored in two unsigned 8 bit color channels. The focus lies on a 16 bit signed number format which generalizes existing 8 bit formats allowing lossless format expansions, and which has an exact representation of 1, 0 and -1 allowing stable long-lasting dynamic texture updates. Implementations of basic arithmetic operations and dependent texture loop-ups in this format are presented and example algorithms dealing with 16 bit precise dynamic updates of displacement maps, normal textures and filters demonstrate some of the resulting application areas.

1 Introduction

The programmability of graphics hardware and the set of available operations has been growing rapidly in recent years. Researches make use of this situation either by extending hardware algorithms and designing new applications which still are supported by new graphics features or by analysing software based graphics packages and extracting or simplifying parts, such that these can be mapped on the new graphics hardware functionality. Both cases have in common that a growing number of complex multi-pass algorithms try to make the best possible use of the available resources and the applications are becoming computationally more and more demanding.

Even when the set of available operations was more restricted various algorithms were designed to exploit graphics features for computations [1, 4, 6]. With the wider availability of extensions like multitexturing and pixel textures the area of applications widened strongly reaching from lighting and shading computations [7, 8, 12, 13] to various image processing applications [5, 9, 10, 11, 20] and advanced hardware accelerated shading languages [14, 15]. The author himself has implemented complicated numerical schemes solving parabolic differential equations fully in graphics hardware [16, 17].

In all these applications dealing with multipasses and multiple textures there is a concern about the inevitably occurring error due to the low resolution of the color channels which usually consist of only 8 bits. Especially when dealing with high dynamic range images [2, 18] several color channels are used together to handle this problem. New HILO texture formats introduced by NVIDIA [3] also mean to provide higher precision texturing in particular for lighting computations. These solutions, however, are restricted to certain applications, so that they cannot be used for arbitrary high precision computations or visualizations. We intend to overcome this difficulty by demonstrating that the existing extensions already allow the introduction of a composite 16 bit format on RGBA8 textures, on which the same operations as on the low precision formats can be applied. The idea behind this emulation, however, is not as much to provide a complete support for 64 bit rendering, as this will

be covered by future graphics hardware more efficiently, but rather to allow an easy and bandwidth efficient concurrent usage of 8 and 16 bit rendering such that the higher precision format can be used to quickly resolve partial accuracy problems on todays 8 bit architectures. Similarly, in the presence of a native 16 bit format this approach would allow to emulate a 32 bit format.

The techniques of implementing high-precision arithmetic from low-precision building-blocks as such are elaborate and have been used in countless architectures. The problem of doing this in graphics hardware, however, lies in the very restricted availability of conditional statements on a per fragment basis. The extensions of pixel shaders and register combiners available on a GeForce3 seemed to offer the highest flexibility for this purpose and have been used for the implementations. Theoretically the alpha-test and some kind of dependent texture access, would also suffice to obtain the same functionality, however, the performance would suffer heavily due to numerous passes. But as the implementations of the emulated operations are independent and do not all rely on the same graphics features, some of them may also be efficiently realized in a more restricted setting.

We will first review the different number formats under OpenGL and explain the choice of a composite 16 bit format. The following main section will then present the implementation of the arithmetic and dependent texture operations and describe example usage.

2 Number Formats

In this section we discuss the different fixed-point number representations in OpenGL and which conditions would be desirable for a new signed 16 bit number format.

Currently standard OpenGL knows only an unsigned 8 bit fixed-point format, but the growing arithmetic within the texture environments tends towards a signed 9 bit format as used by the register combiners. Recently NVIDIA introduced a signed 8 bit format and a signed and unsigned 16 bit format. Unfortunately switching from a lower to a higher precision format does not always imply that all numbers in the lower precision format can be exactly represented in the higher one. This problem is not specific to the OpenGL setting, but a general difficulty in defining fixed-point representations. The situation is even more confusing as the unsigned formats may be re-interpreted as signed numbers by the mapping $x \mapsto 2x - 1$, which is available at some stages in the graphics pipeline. Table 1 gives an overview of the different formats.

We see that the unsigned 8 bit format represents a subset of the unsigned 16 bit format and that the signed 8 bit represents a subset of the signed 16 bit format, so that these pairs of formats can be used together effectively. But the represented numbers from the signed and unsigned formats have almost nothing in common, so that conversions between these would inevitably lead to loss of precision, which would prohibit such conversions in accumulating texture updates. Therefore we will require the new composite format to be a superset of all of the lower precision formats. Naturally the other 16 bit formats cannot be generalized by a format with the same resolution.

The best generalization so far of signed and unsigned formats is given by the signed 9 bit format which is a superset of both the unsigned 8 bit and the normal expansion of the unsigned 8 bit format. Moreover, the signed 9 bit format has the advantage the it exactly represents the neutral elements of addition 0 and multiplication 1 and its divisors -1, which is very important for long-lasting dynamic texture updates. If, for example, we dynamically change a texture every other frame using additions and multiplications, but want some areas of the texture to remain unchanged for some time or even throughout the process, we must rely on the exact representation of 0 and 1 or else we would have to store the information about every region to be protected somewhere and use some sort of fragment test to leave them unchanged. By generalizing the singed 9 bit format as required above, we will automatically transfer this property of exact representation of $\{0, 1, -1\}$ to the composite format.

Additionally we would want the first 8 bit color channel of the composite 16 bit number representation to be - on its own - the best possible approximation of the signed 16 bit number. This would allow us to use only the first part in cases where the full precision is not required or difficult to use. Finally we should choose a format which requires only few operations to perform the carry-over arithmetic necessary for 16 bit operations performed on signed 8 bit multipliers and adders. Thus, we may

Table 1: Comparison of number formats in OpenGL.

	unsigned 8 bit	unsigned 16 bit	unsigned 8bit $x \mapsto 2x-1$	signed 9 bit	signed 8 bit	signed 16 bit
formula	$a/255$	$a/(255 \cdot 257)$	$(2a-1)/255$	$a/255$	$a/128$	$a/(128 \cdot 256)$
range of a	$[0,255]$	$[0,65535]$	$[0,255]$	$[-256,255]$	$[-128,127]$	$[-32768,32767]$

summarize the conditions for the desired signed 16 bit format as follows:

- The composite number format should be a superset of all lower precision formats from Table 1.
- Its first 8 bit channel should be the best possible approximation to the whole signed 16 bit number.
- The format should allow an efficient implementation of the carry-over arithmetic.

A format which fulfills these conditions can be defined on a RGBA8 texture in the following way. We let (R, A) represent the first signed 16 bit number and (G, B) the second. The representation of a fixed-point number is given by:

$$\begin{aligned} \psi(r,a) &:= \tfrac{1}{255}\left((2r-255) + \tfrac{1}{128}(a-128)\right) \\ &= \tfrac{1}{255\cdot 128}\left(256(r-128) + a\right) \\ \Psi(R,A) &:= \left(2R-1\right) + \tfrac{1}{128}\left(A - \tfrac{1}{2}\right) \\ &= \left(2R - \tfrac{256}{255}\right) + \tfrac{A}{128}. \end{aligned}$$

The first rows define the correspondence for integer r and a, and the second for fixed point $R = r/255$ and $A = a/255$ (1 corresponds to 255 and $\frac{1}{2}$ to 128). From the first formula we see that our composite format generalizes the lower precision formats from Table 1, as it produces all numerators from $-255 \cdot 128$ to $+255 \cdot 128$ for the common denominator $255 \cdot 128$. Moreover, by definition the mapping of the first color channel R corresponds to the normal expansion of an unsigned 8 bit format giving the best possible approximation to the whole 16 bit number. Thus the format fulfills the first two conditions given above. The satisfaction of the third condition will become clear in Section 3 where we will present the exact implementations of arithmetic operations.

At the end of this section we should look at possible drawbacks of this format. The problem with fixed-point numbers having an exact representation of $\{0, 1, -1\}$ is an unavoidable representation of numbers outside of the range $[-1, 1]$. In case of the signed 9 bit format this is $-\frac{256}{255}$ and in our case all numbers with $r = 255, a > 128$ and $r = 0, a < 128$. It would be very unpleasant having to define an external format with a resolution depend number range. This problem is well known and has influenced the decision against such external formats as discussed in NVIDIA's texture shader specification [3]. But as in the case of the signed 9 bit format additional clamping operations would solve the main disadvantage of over-representation and would gain smoother transitions between the existing formats.

3 Operations

In this section we will present how the basic arithmetic operations addition, subtraction, multiplication, division and dependent texture look-ups can be realized at 16 bit precision with our composite number format in RGBA8 textures. Short examples will demonstrate possible uses of the operations.

While the predefined 16 bit formats can only be used in few special operations and their values must be uploaded from main memory, the operations from this section will exhibit the main advantage of the new composite format by allowing dynamic changes to the the operands.

All occurring textures will be two dimensional. They will usually contain the first 16 bit channel in the colors (R, A) and the second in (G, B), where $R, G, B, A \in [0, 1]$ represent fixed-point values in 8 bit. This choice will become clear in Section 3.3. Textures will be seen as two-dimensional four-valued mappings: $D : (x, y) \longmapsto D[RGBA](x, y)$. The necessary operations in the pixel shaders and register combiners will be given in pseudo-code notation, where '$\longmapsto$' means *is mapped to* , '$\rightarrow$' means *is stored in* , '$\bullet$' denotes the dot-product and '$\cdot$' the component-wise multiplication.

3.1 Dependent Texture Look-Ups

Let T be any texture which should be accessed via a dynamically generated displacement map D with the 16 bit x-displacement in (R, A) and the 16 bit y-displacement in (G, B). Then a quick 8 bit precise texture offset $T(x + s \cdot D[R], y + s \cdot D[G])$ can be realized in pixel shaders through:

$\mathbf{T(x + s \cdot D[R], y + s \cdot D[G])}$

0: tex2d	$(x, y, z) \longmapsto D(x, y)$
1: dot2d blue to 1	$u_x = (2s, 0, x - s) \bullet (D[R], D[G], 1)$ $= x + s \cdot (2D[R] - 1)$
2: lut2d blue to 1	$u_y = (0, 2s, y - s) \bullet (D[R], D[G], 1)$ $= y + s \cdot (2D[G] - 1)$
	$\longmapsto T(u_x, u_y)$

where s is a user set scaling factor, which determines the maximal possible offset.

Example: Brownian motion. Let V be a vector field with random x-components in (R, A), random y-components in (G, B) and the wrap mode set to repeat, and D a displacement map as above initially set to zero. Then the following short algorithm will produce a random local motion in the texture T:

$r_x = \text{random}(2^{16})$;
$r_y = \text{random}(2^{16})$;
$V = V(x + r_x \cdot V[R], y + r_y \cdot V[G])$;
$D \mathrel{+}= \tau \cdot V$;
$R = T(x + D[R], y + D[G])$;

where τ steers the speed of the motion and R holds the resulting texture in each step. Although the look-up itself takes place in only 8 bit, the motion can vary across the texture with 16 bit since the displacement D is stored and calculated in 16 bit. The implementation of 16 bit precise addition and multiplication is shown in the following subsections. In particular, the user may vary τ within a bigger range, without having to fear that the motion stops altogether, because the multiplication with τ evaluates to zero.

As the predefined texture offset operations require a DSDT or HILO input format we would currently need two separate textures for x-displacement $D_x[GB]$ and y-displacement $D_y[GB]$ with $D_x[R]$ and $D_y[R]$ set to one for a dynamic 16 bit precise look-up in pixel shaders:

$\mathbf{T(x + s \cdot D_x[GB], y + s \cdot D_y[GB])}$

0: tex2d	$(x, y, z) \longmapsto D_x(x, y)$
1: tex2d	$(x, y, z) \longmapsto D_y(x, y)$
2: dot2d	$u_x = (x + s\frac{256}{255}, 2s, s/128)$ $\bullet (1, D_x[G], D_x[B])$ $= x + s \cdot \Psi(D_x[G], D_x[B])$
3: lut2d	$u_y = (y + s\frac{256}{255}, 2s, s/128)$ $\bullet (1, D_y[G], D_y[B])$ $= y + s \cdot \Psi(D_y[G], D_y[B])$
	$\longmapsto T(u_x, u_y)$

where s again scales the offset. Here we could also obtain an absolute and not an offset-texture look-up by eliminating the coordinates x and y in the pixel shaders 2 and 3. Then D_x, D_y would address T absolutely, i.e. the result would be $T(D_x[GB], D_y[GB])$. Such a construction can be used to evaluate an arbitrary function f of two 16 bit variables $f(X, Y) = T(X[GB], Y[GB])$. The precision of this evaluation corresponds directly to the size of the texture T which holds the resulting values. In particular we could implement a *division* between two 16 bit numbers in this way, but naturally the resulting range would still be confined to the same interval for all pixel values in an image, i.e. division by small numbers really requires floating-point formats.

We should also emphasize that the above use of the dot-product 2d operation, where the first part (shader 2) accesses a different previous texture, namely D_x, than the second D_y, is uncommon but legitimate.

Example: Advection. Let $V_x[GB]$ be the x-component and $V_y[GB]$ the y-component of a continuous vector field V. Alike let $D_x[GB]$ and $D_y[GB]$ be the x and y-components of a displacement map initially set to zero. Then the following short algorithm will produce an advection of the texture T along the vector field V:

$D_x \mathrel{+}= \tau \cdot V_x(x - \tau D_x[GB], y - \tau D_y[GB])$;
$D_y \mathrel{+}= \tau \cdot V_y(x - \tau D_x[GB], y - \tau D_y[GB])$;
$R = T(x - D_x[GB], y - D_y[GB])$;

where τ again steers the speed of the motion and R holds the resulting advected texture in each step. If we replaced V_x by D_x and V_y by D_y we would obtain a self-advection of D_x, D_y, which is a step towards fluid dynamics. In this manner we could simulate the motion of gas or water if we also

forced (D_x, D_y) to be divergence free as required by the incompressible Navier-Stokes-Equations, for more details we refer to [19].

3.2 Addition and Subtraction

Let the texture $tex0$ hold a pair of 16 bit precise x and y coordinates in $(tex0[R], tex0[A])$ and $(tex0[G], tex0[B])$ respectively, and the texture $tex1$ another pair in $(tex1[R], tex1[A])$, $(tex1[G], tex1[B])$. For the x and y coordinate let $(s_x, s_y) \in \{-1, 1\} \times \{-1, 1\}$ encode independently whether an addition (1 set) or a subtraction (-1 set) should be performed. Then the two simultaneous 16 bit precise additions or subtractions of the x and y coordinates can be mapped onto the register combiner functionality:

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

The result of the two parallel 16 bit precise additions or subtractions lies in $tex0$ and can be further processed by more register combiners or lighting operations in the final combiner. For clarity of presentation input mappings for constants are not explicitly given, but the ranges of the color channels have been chosen such that there always exists an appropriate mapping. Moreover, due to the number of occurring constants we use the NV_register_combiners2 extension providing combiner dependent constants.

Each 16 bit addition above is emulated by performing a componentwise addition on the color channels, then checking the sum of the lower component for an overflow and finally correcting both components appropriately. The *great advantage* of the introduced composite number format is that only one such check is necessary to handle both positive and negative overflow for both addition and subtraction. In this way our format fulfills the third condition required in Section 2. This efficient carry-over arithmetic is due to the fact that the initial input mapping $x \mapsto x - \frac{1}{2}$ for the addends is not the correct mapping ($x \mapsto 2x - 1$) for the first color channel in our representation. The resulting difference introduces a $\frac{s}{255}$ error, which cannot be represented in the first color channel. But the appropriate $\frac{1}{255}$ correction applied to the second color channel corrects a possible overflow therein and the sum of these corrections can be represented in the first channel. Therefore only one condition has to be checked to decide in which direction a correction on the second channel should take place.

The reason for the awkward repeated negation in combiners 1[RGB], 2[RGB] and similar in 3[RGB], 4[RGB] and 4[A] with intermediate $(-\frac{1}{2})$-output mapping is an effort to implicitly realize an addition of $\frac{1}{2}$ although there is no such input or output mapping. It is necessary for the re-encoding of the results from the signed range $[-\frac{1}{2}, \frac{1}{2})$ back to $[0, 1]$.

Example: Rotation of normals. Let the texture N_x define the x-component and the texture N_y the y-component of a normal map, whereas the z-component is implicitly defined if we think of the normals to be of unit length. Then we can use the following algorithm to rotate the normals around the z-axis.

$$N_x = (1 - c)N_x + cN_y;$$
$$N_y = (1 - c)N_y - cN_x;$$
$$R = F(N_x[G], N_y[G])$$

where c is a constant steering the speed of the rotation and F is a texture which, addressed by the main components of N_x and N_y, delivers the normal with the computed z-component $\sqrt{1 - N_x[G]^2 - N_y[G]^2}$.

3.3 Functions and Multiplication

We have suggested to arrange the two 16 bit numbers into the color channel pairs (R, A) and (G, B) because these pairs can be used for a dependent texture look-up. Such look-ups implement the application of arbitrary functions on our composite 16 bit format and within the 4 pixel shaders both 16 bit channels can be mapped by a different function. Let D be again a displacement map with an x and y component as before, and let F_x and F_y be 256×256 textures encoding nonlinear functions on the composite 16 bit format. Then we can apply F_x and F_y simultaneously to D using pixel shaders:

$\mathbf{F_x(D[RA])},\ \mathbf{F_y(D[GB])}$

0: tex2d $(x, y, z) \longmapsto D(x, y)$

1: ar2d $(D[A], D[R]) \longmapsto F_x(D[RA])$

2: gb2d $(D[G], D[B]) \longmapsto F_y(D[GB])$

The textures F_x and F_y should contain the values such that addressed by AR, where R is the first color channel in the number representation, they deliver the function value in AR and addressed by GB they deliver it in GB. If it is clear that a function need not to be used in both dependent modi, then a single texture F storing the values of both F_x and F_y would suffice. One could evade this difficulty by storing the first channel of the number representation in A instead of R, but this would imply many more difficulties in other operations.

Example: Linear filters. In the former examples we have used multiplications of the form $\tau \cdot V$ where τ is a user defined constant and V an intermediate texture result. To implement such an multiplication in 16 bit precision one defines a texture T_τ containing the product values of arbitrary 16 bit values with τ and applies it to V obtaining $(T_\tau[RA](D[RA]), T_\tau[GB](D[GB]))$. Since the application of the function uses only the pixel shaders and the addition only the register combiners, one can perform an operation like $D + \tau \cdot V$ in one pass. In particular one can quickly implement a 16 bit precise filter using a 3 by 3 stencil:

$$R = \sum_{d_x, d_y = -1}^{1} c_{d_x, d_y} \cdot T(x + d_x, y + d_y);$$

where c_{d_x, d_y} are the filter coefficients and R contains the filtered texture T. If each of the coefficients is different, which is seldom the case, one would need 9 textures T_{d_x, d_y} encoding the values of a multiplication with c_{d_x, d_y} and also 9 passes to compute the result R. But as all computations are performed in 16 bit, the result is significantly better than in 8 bit, especially for small coefficients.

In terms of hardware resources, the multiplication is a much more complex operation than the addition and therefore more difficult to emulate using lower precision computing blocks. The starting point is the decomposition of the 16 bit product into a sum of 8 bit products. Let C and D be two textures encoding the 16 bit numbers to be multiplied in the colors (G, B). First multiplying the representations of $C[GB]$ and $D[GB]$ we obtain:

$$\begin{aligned}
&\Psi(C[G], C[B]) \ \cdot\ \Psi(D[G], D[B]) = \\
&\quad \big(2C[G] - 1\big) \cdot \big(2D[G] - 1\big) \\
&\quad + \tfrac{1}{128} \Big(\big(2C[G] - 1\big) \cdot \big(D[B] - \tfrac{1}{2}\big) \\
&\qquad + \big(2D[G] - 1\big) \cdot \big(C[B] - \tfrac{1}{2}\big) \Big) \\
&\quad + \tfrac{1}{128^2} \Big(\big(C[B] - \tfrac{1}{2}\big) \cdot \big(D[B] - \tfrac{1}{2}\big) \Big)
\end{aligned}$$

The first addend of the result can be evaluated at 16 bit to $M(C[G], D[G])$ by a texture look-up with the first components $C[G]$ and $D[G]$ addressing a multiplication table M. The second addend may be computed and rounded by the register combiners, while the third gives at most $\frac{1}{256^2}$, which is less than one half of the smallest representable number $\frac{1}{255 \cdot 128}$, and thus will be ignored. In this way we can implement a one-pass texture-texture multiplication in 16 bit precision, but unlike the addition only one of the 16 bit channels can be multiplied at once.

$\mathbf{C[G] \cdot D[G]}$

0: tex2d $(x, y, z) \longmapsto C(x, y)$

1: tex2d $(x, y, z) \longmapsto D(x, y)$

2: dot2d $u_x = (0, 1, 0) \bullet (0, C[G], C[B])$
$= C[G]$

3: lut2d $u_y = (0, 1, 0) \bullet (0, D[G], D[B])$
$= D[G]$

$\longmapsto M(u_x, u_y)$

The resulting textures $tex0, tex1$ and $tex3$ are now further processed in the register combiners to compute the mixed products and sum up the addends of the multiplication formula.

$\mathbf{tex3[GB]+}$ $\mathbf{tex0[G]\cdot tex1[B]+tex1[G]\cdot tex0[B]}$	
0: RGB	$(2\cdot tex0[RGB]-1)\bullet(0,1,0)$ $\rightarrow\ sp0[RGB]$, $(2\cdot tex1[RGB]-1)\bullet(0,1,0)$ $\rightarrow\ sp1[RGB]$
1: A	$sp0[B]\cdot(tex1[RGB]-\frac{1}{2})\ +\ sp1[B]\cdot$ $\cdot\ (tex0[RGB]-\frac{1}{2})\ \rightarrow\ sp0[A]$
2: RGB	$(0,0,1)\cdot sp0[A]\ \rightarrow\ tex0[RGB]$
2: A	$sp0[A]\ +\ \frac{1}{2}\ \rightarrow\ sp0[A]$
3: RGB	$(sp0[A]<\frac{1}{2})?\ -(0,-1/2^8,\frac{1}{2})$ $:\ -(0,0,-\frac{1}{2})$ $\xrightarrow{-\frac{1}{2}}\ sp0[RGB]$
4: RGB	$tex0[RGB]\ +\ (0,-1,-1)\ \cdot$ $\cdot\ sp0[RGB]\ \rightarrow\ tex0[RGB]$
5: RGB	$(tex0[RGB]-\frac{1}{2})\ +$ $(tex3[RGB]-\frac{1}{2})\ \rightarrow\ tex0[RGB]$
5: A	$tex0[B]+(tex3[B]-\frac{1}{2})\ \rightarrow\ sp0[A]$
6: RGB	$(sp0[A]<\frac{1}{2})?\ -(0,0,\frac{1}{2})$ $:\ -(0,1/2^8,-\frac{1}{2})$ $\xrightarrow{-\frac{1}{2}}\ sp0[RGB]$
7: RGB	$tex0[RGB]\ +\ (0,-1,-1)\ \cdot$ $\cdot\ sp0[RGB]\ \rightarrow\ tex0[RGB]$

The main calculations take place in combiner 1[A] where the sum of the mixed products is computed, and in combiner 5[RGB] where the former sum is added to the result of the multiplication table. Combiners 3[RGB], 4[RGB] and 6[RGB], 7[RGB] perform again the carry-over arithmetic as in the case of addition and subtraction. They have different correction directions, because of the implicit 0 in tex0[G] due to combiner 2[RGB].

Example: Nonlinear filters. In the last example we have seen how a texture can be filtered with a stencil of constant coefficients. If we use several textures instead of the constants, the coefficients may vary depending on the coordinates and we obtain a nonlinear filter:

$$R=\sum_{d_x,d_y=-1}^{1} W^{d_x,d_y}(x,y)\cdot T(x+d_x,y+d_y);$$

where W^{d_x,d_y} are now textures containing the varying weights of the filter for each direction (d_x,d_y). Nonlinear filters can be effectively used for edge sensitive denoising of images. Figure 1 shows the advantages of the increased precision in this application.

3.4 Performance

Apart from the multiplication the new operations on the composite 16 bit format will perform at almost 50% of the normal speed. Using the dot-product operation instead of the offset-texture for dependent texture look-ups costs a factor of 2. The 5 combiners of the addition would normaly cost a factor of 3, but since some sort of the much slower dependent texture access will usually preceed the addition (a multiplication with a constant for example) multiple register combiners seldom reduce overall performance. Finally, the multiplication is comparably slow because the dot-product operation takes 8 times longer than a normal texture access. This factor, however, is not surprising as the complexity of a multiplication grows quadratically with the bitlength of the operands, so it is rather amazing that it can be realized in a single pass at all. Moreover, other time consuming procedures such as texture object switching or implicit pipeline flushing may absorb these theoretical extra costs, as has been experienced in the filter example (Figure 1).

4 Conclusions

A composite 16 bit number format has been presented on which precise arithmetic and dependent texture operations can be efficiently performed. In particular this format allows dynamic accurate changes to displacement maps, normals and filters. These high precision operations naturally require more texture memory and computing time, but are still fast enough to be used in precision sensitive parts of real-time multi-pass algorithms. Also the details of this 16 bit emulation may seem deterrent at first, however, once implemented the operations can be used in a simple modular way. We hope that by use of this virtual 16 bit format more precision sensitive visualization and computing can be accelerated in graphics hardware.

References

[1] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic

reconstruction using texture mapping hardware. In Arie Kaufman and Wolfgang Krueger, editors, *1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, 1994. ISBN 0-89791-741-3.

[2] Jonathan Cohen, Chris Tchou, Tim Hawkins, and Paul Debevec. Real-time high dynamic range texture mapping. In *Proceedings of the Eurographics Rendering Workshop 2001*, 2001.

[3] NVIDIA Corporation. NVIDIA OpenGL extension specifications. Technical report, NVIDIA Corporation, 2001.

[4] Paul J. Diefenbach and Norman I. Badler. Multi-pass pipeline rendering: Realism for dynamic environments. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 59–70. ACM SIGGRAPH, 1997. ISBN 0-89791-884-3.

[5] U. Diewald, T. Preusser, M. Rumpf, and R. Strzodka. Diffusion models and their accelerated solution in computer vision applications. *Acta Mathematica Universitatis Comenianae (AMUC)*, 70(1):15–31, 2001.

[6] Paul Haeberli and Mark Segal. Texture mapping as A fundamental drawing primitive. In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *Fourth Eurographics Workshop on Rendering*, pages 259–266. Eurographics, June 1993. held in Paris, France, 14–16 June 1993.

[7] 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.

[8] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In Alyn Rockwood, editor, *Siggraph 1999, Annual Conference Proceedings*, Annual Conference Series, pages 171–178. ACM Siggraph, Addison Wesley Longman, 1999.

[9] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics*, 33(Annual Conference Series):277–286, 1999.

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

[11] M. Hopf and T. Ertl. Hardware Accelerated Wavelet Transformations. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 93–103, 2000.

[12] Jan Kautz and Michael D. McCool. Interactive rendering with arbitrary BRDFs using separable approximations. In ACM, editor, *SIGGRAPH 99. Proceedings of the 1999 SIGGRAPH annual conference: Conference abstracts and applications*, Computer Graphics, pages 253–253. ACM Press, 1999.

[13] Michael D. McCool and Wolfgang Heidrich. Texture shaders. In ACM, editor, *SIGGRAPH '99. Proceedings 1999 Eurographics/SIGGRAPH workshop on Graphics hardware*, Computer Graphics, pages 117–126. ACM Press, 1999.

[14] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings,*, Annual Conference Series, pages 425–432. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[15] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 159–170. ACM Press / ACM SIGGRAPH, 2001.

[16] M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *Proceedings ICIP'01*, volume 3, pages 1103–1106, 2001.

[17] M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings VIIP'01*, pages 193–202, 2001.

[18] A. Scheel, M. Stamminger, and H.-P. Seidel. Tone reproduction for interactive walkthroughs. *Computer Graphics Forum*, 19(3), August 2000.

[19] Jos Stam. A simple fluid solver based on the FFT. *Journal of Graphics Tools*, 6(2):43–52, 2002.

[20] Chris Trendall and A. James Stewart. General calculations using graphics hardware, with applications to interactive caustics. In *Eurographics Workshop on Rendering*, June 2000.

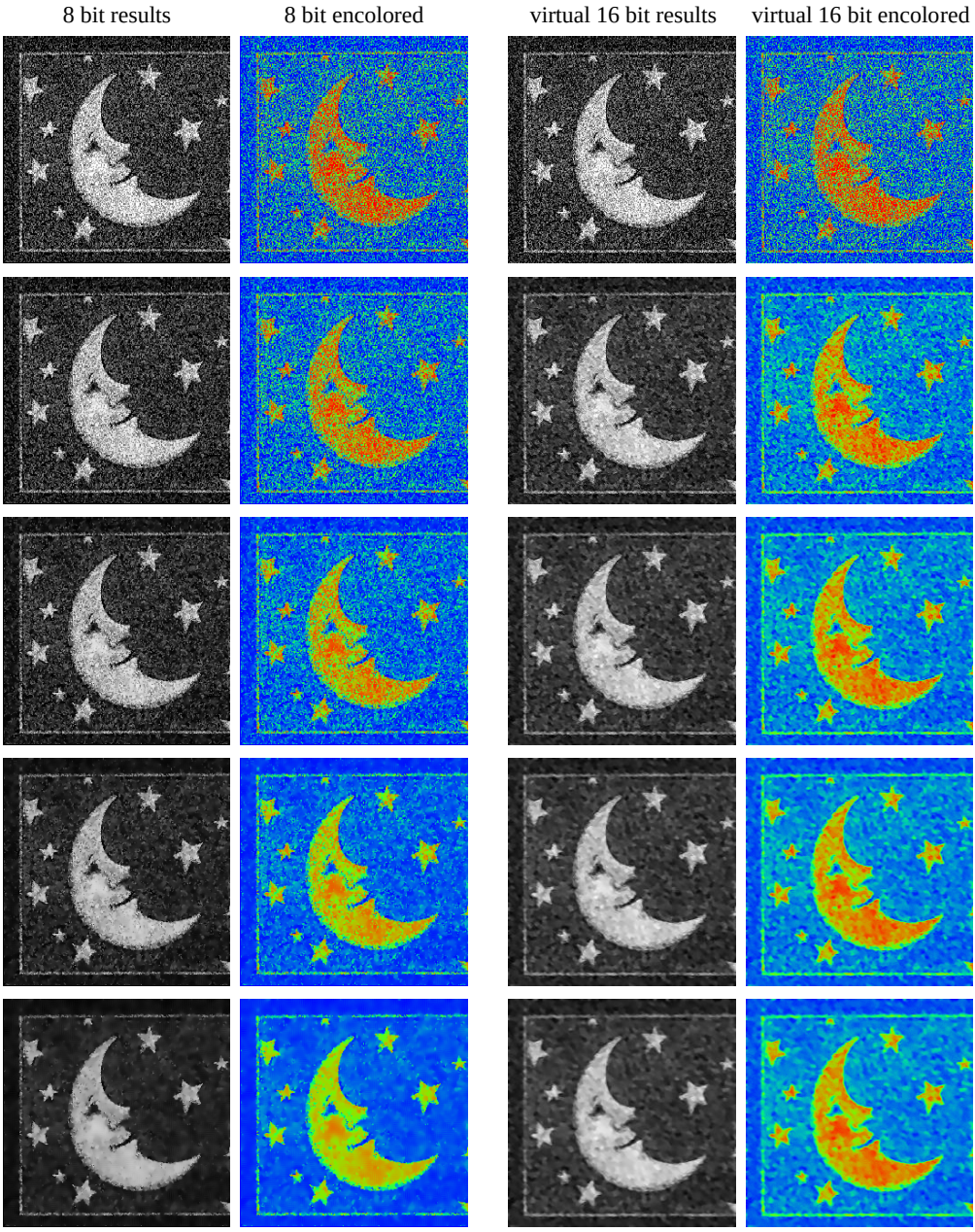

Figure 1: From top to bottom every tenth result of a nonlinear diffusion filter applied to a noisy 256^2 image is shown. Although the last 8 bit result may seem pleasant at first, the dark blue background and the yellow and green color of the moon clearly convey a mass defect. The new 16 bit format, on the other hand, preserves the overall mass and eliminates artefacts much smoother due to the finer quantization.
For a direct comparison both sequences were computed on RGB8 textures (the read-back for LA8 was very slow). The 8 bit computation took 5ms for a time-step and the virtual 16 bit computation 8ms. Although three independent 8 bit filters could have been used in parallel on RGB8, the performance of more than 50% of the normal speed together with the higher quality results count in favour of the virtual 16 bit format.