Per-pixel divisions

Sergey Parilov   Wolfgang Stuerzlinger
Department of Computer Science
York University
4700 Keele Street, M3J 1P3
Toronto, Ontario, Canada

{parilov|wolfgang}@cs.yorku.ca

Abstract

We describe a method to perform per-pixel divisions using graphics hardware, with a simple iterative algorithm. In the worst case, the number of rendering passes is 17 for approximate divisions, and 18 for accurate divisions. The algorithm can be used for 3D vector normalization, evaluating complex lighting models, and image reconstruction.

1 Introduction and Previous Work

Graphics hardware can act as a powerful SIMD data processing unit. However, the numerical precision is limited to the available number of bits per color channel of a pixel, and only fairly simple operations are available in the hardware. In particular, the OpenGL standard contains no operation for a single per-pixel division operation [3]. This paper presents an algorithm for computing division using multiple rendering passes.

Alternatives to our division algorithm,described below, could be using look-up tables storing $\frac{1}{Alpha}$ values or using pixel textures to perform R,G,B,A to R,G,B look-up's [3]. Unfortunately, even 16-bit look-up tables result in considerable loss of precision, when set-up to evaluate $\frac{1}{x}$. Pixel textures are not widely supported by the graphics hardware and have an additional drawback of consuming too much memory.

One of our applications of the division algorithm is image reconstruction from non-uniformly distributed samples, which requires the computation of a weighted average of the form $\frac{\sum_i w_i \cdot I_i}{\sum_i w_i}$ per pixel [2].

The division could also be used to perform per-pixel vector normalization. If the coordinates of a 3D-vector are stored in R,G,B components of every pixel, it is possible to use blending operations and color look-up tables to compute the length of the vector in the alpha channel. Then we normalize the vector by dividing each component by its length, which can be used for realistic shading. The same result can be achieved with the parabolic and cube maps for 3D vector normalization.

The following section describes how we use the OpenGL functions mentioned above to perform the division using the general graphics hardware.

2 Performing the division

We use an iterative approach to compute the result x of the division

$$x = \frac{y}{z} = \frac{x \cdot z}{z} \tag{1}$$

for some $x, y, z \leq 1$. This choice is dictated by the fact that current implementations of OpenGL operate on values in the range of $[0 \ldots 1]$.

To find x given $y = x \cdot z$ and z, we find a set of scaling factors s_i, between 1 and 2, such that $z \cdot s_0 \cdot \ldots \cdot s_n \approx 1$. Then, $s_0 \cdot \ldots \cdot s_n \approx \frac{1}{z}$ and $y \cdot s_0 \cdot \ldots \cdot s_n = x \cdot z \cdot s_0 \cdot \ldots \cdot s_n \approx x \cdot z \cdot \frac{1}{z} = x$.

From here on, we assume that the Red channel of the frame buffer initially stores the values y, and the Alpha channel initially stores the values z, for every pixel. Denote these *Red* and *Alpha* respectively.

Our algorithm consists of multiple steps. At each step i we are going to set s_i so that *Alpha* increases gradually to 1, while avoiding overflow in both *Red* and *Alpha*.

At step i, *Red* and *Alpha* contain $y \cdot s_0 \cdot \ldots \cdot s_i$ and $z \cdot s_0 \cdot \ldots \cdot s_i$ respectively. At each step, the values *Red* and *Alpha* grow by a little as the result of multiplication by s_i. At the end, *Red* will contain x, and *Alpha* will contain 1, which is the desired result.

At each step, all the $Alpha_f$ values are greater than some value low. We continuously raise this lower boundary, until low becomes close to 1. Our algorithm consists of two phases. In the first phase, we raise low so that it is greater than or equal to 0.5. In the second phase, we raise low from 0.5 to 1.

At the first step, we assume $low = \frac{1}{255}$ (when using 8 bits to represent intensities)[1]. We can easily raise the lower boundary to 0.5 by successively multiplying all the values that are less than 0.5 by a factor of 2. This constitutes the first phase of our division algorithm. In this phase, we are only working with the pixels with $Alpha < 0.5$. In the worst case, it takes no more than 7 iterations over the image, since $2^7 \frac{1}{255} > 0.5$. Using blending and alpha test, we effectively run the pseudo-code presented in the figure 1.

```
for iteration=1..7
  for all pixels in the image, in parallel
    if(alpha<0.5)
      alpha=alpha+alpha
      red=red+red
```

Figure 1: The first phase of the division algorithm.

After the first phase, we can assume that the lowest alpha value in the frame buffer (low) will be greater than 0.5. At this point, we cannot continue multiplying by 2, since it will result in values greater than 1, which will be clamped by OpenGL.

To raise the lower boundary, we then pick an interval $[low_i, high_i]$ of alpha values for the step i. All the alphas within this range will be multiplied by some constant c_i.

low_i is our current smallest alpha value in the frame buffer. Our choice of $high_i$ and c_i is determined by the following factors -

- $c_i > 1$, since we want to raise our lower boundary low;
- $high_i * c_i = 1$, since we do not want any overflow in the frame buffer;
- $low_i * c_i = low_{i+1}$, since the lowest alpha value possible in step $i + 1$ is the lowest value for step i, multiplied by c_i;
- $low_{i+1} = high_i$, since we want to to perform as few multiplications as possible.

[1] $Alpha = 0$ corresponds to a division by 0. Our algorithm will produce arbitrary results in this case.

Solving the above system of constraints for c_i and $high_i$ yields

$$c_i = \sqrt{\frac{1}{low_i}}, \tag{2}$$

and

$$high_i = low_i * c_i = \sqrt{low_i}. \tag{3}$$

As mentioned before, there is no way to perform a per-pixel division in a few simple OpenGL operations. Moreover, there is no easy way to perform per-pixel multiplications by a factor bigger than 1 without considerable loss of precision, which could be used to replace divisions by some a with multiplications by $\frac{1}{a}$. Nevertheless, we can imitate a multiplication by some factor a between 1 and 2 by using the equivalence $ax = x + (a - 1)x$. This can be easily computed using the blending operation.

Given the equations for c_i and $high_i$, we iterate over the image increasing the low_i value at every step until it becomes 1. This constitutes the second phase of our division algorithm. Figure 2 presents the pseudocode.

```
low = 0.5

for iteration = 1..9
  c=sqrt(1/low) /* computed on the CPU */
  high=low*c

  for all pixels in the image, in parallel
    if(alpha<=high)
      alpha=alpha+(c-1)*alpha
      red=red+(c-1)*red

  low=high
```

Figure 2: The second phase of the division algorithm.

Table 2 shows the convergence of the division algorithm. The entire second phase requires a maximum of 9 iterations over the image. As the table indicates, subsequent iterations would involve values of $c_i - 1$ below the frame-buffer precision, and do not influence the result. Since the

OpenGL standard [3] does not specify how exactly the arithmetic operations are performed, we assume that each intermediate result is rounded down to the closest representable number, which is the worst case for our algorithm. 12 bits per channel in the frame buffer provide enough precision to perform the computations. When working with 8-bit frame buffers, the algorithm converges after 8 iterations with a value of 253 in the alpha channel. Note that better results can be expected on hardware that properly rounds the intermediate results.

When additional precision is desired, the values that are not fully normalized ($Alpha = 254$) can be updated with an additional rendering pass. One would need to set up the look-up table for the red channel to compute $Red = Red \cdot \frac{255}{254}$. Then, this look-up should be performed for all pixels with $Alpha = 254$, using alpha tests to operate only on the desired subset of pixels.

To test our algorithm, we performed all valid divisions with 8 bit arguments and 8 bit result, for $Red, Alpha \leq 1$. In all cases, the result was either the correct value, or the value one greater than the correct one. For example, $0.22/0.77 \approx 0.29$, expressed 8-bit fixed point $56/196 = 72$. Our algorithm returns 73 in this case. This deviation from the ideal result occurs in 38% of all possible 8 bit divisions.

i	c_i	low_i	$high_i$	$\lfloor low_i * 255 \rfloor$
1	1.4142	0.5000	0.7070	127
2	1.1893	0.7070	0.8408	180
3	1.0906	0.8408	0.9167	214
4	1.0444	0.9167	0.9573	233
5	1.0221	0.9573	0.9783	244
6	1.0110	0.9783	0.9890	249
7	1.0055	0.9890	0.9944	252
8	1.0028	0.9944	0.9971	253
9	1.0015	0.9971	0.9983	254
10	1.0009	0.9983	0.9990	254

Table 1: Worst case convergence of the second phase of the division algorithm. All computations are performed with 12-bit precision, with hardware that truncates the intermediate results. (See text for how to get correct result.)

3 Discussion and Conclusions

The described division with the help of graphics hardware is an example of multi-pass rendering. We need one pass to upload the image to the graphics hardware, 7 passes to complete the first phase of our division algorithm, and 9 passes to complete the second phase. In total, we make 17 rendering passes, which is not a problem given the pixel fill-rates of the modern hardware. If lower precision is acceptable, we can stop iterations of the second phase of the algorithm earlier. Also, a better estimate of the smallest denominator in the frame buffer can reduce the number of iterations in the first phase of our division algorithm.

Recent graphics hardware, i.e. released during 2002, offers the ability to do dependent texture lookups. This can be used to do three texture lookups (one for each color channel) on a two-dimensional table of pre-computed values y/z [2][1]. Subsequent generations of graphics hardware will offer floating point frame buffers with support for arbitrary per pixel computations.

Our division algorithm does not require any special graphics hardware capabilities beyond alpha blending, although we need OpenGL GL_ARB_imaging extension for the second phase of our algorithm. Moreover, the algorithm can also be applied on framebuffers with more than 8 bits. For example, on a 16 bit framebuffer, the second phase will finish after 17 iterations (if stopped after 9 iterations, the error will be less than 0.2%). The whole algorithm will compute the correct result in 25 passes. The actual C/OpenGL implementation of the algorithm is presented in the figure 3.

References

[1] nVidia, *Overview of graphics hardware*, 2002.

[2] S. Parilov, *Real-time rendering of large point-sampled scenes*, Master's thesis, York University, July 2002.

[3] M. Segal and K. Akeley, *The OpenGL (R) Graphics System: A Specification (Version 1.3)*, Copyright ©1992-2001 Silicon Graphics, Inc.

[2]Pointed out by an anonymous reviewer.

```
/* for each pixel R,G,B channels contain the numerators,
   Alpha channel contains the denominators,
   assume alpha-blending and alpha-tests enabled */

glRasterPos2i(0,0);

/* the first phase */
glAlphaFunc(GL_LEQUAL, 0.5);
glBlendFunc(GL_ONE, GL_ONE);

for(i=0;i<7;i++)
  glCopyPixels(0,0,WIN_SIZE_X,WIN_SIZE_Y,GL_COLOR);


/* the second phase */
glBlendFunc(GL_CONSTANT_ALPHA, GL_ONE);

low=0.5;

for(i=0;i<9;i++)
  {
    c=sqrt(1/low);
    high=low*c;

    glAlphaFunc(GL_LEQUAL, high);
    glBlendColor(c,c,c,c);
    glCopyPixels(0,0,WIN_SIZE_X,WIN_SIZE_Y,GL_COLOR);

    low=high;
  }

/* at this point R,G,B channels of each pixel in the frame buffer
   contain the results of the division, Alpha channel is set to 1 */
```

Figure 3: The actual C/OpenGL code performing the division.