Algorithms for Division Free Perspective Correct Rendering

B. Barenbrug, F.J. Peters, and C.W.A.M. van Overveld*

Philips Research Laboratories

Abstract

Well known implementations for perspective correct rendering of planar polygons require a division per rendered pixel. Such a division is better to be avoided as it is an expensive operation in terms of silicon gates and clock cycles. In this paper we present a family of efficient midpoint algorithms that can be used to avoid division operators. These algorithms do not require more than a small number of additions per pixel. We show how these can be embedded in scan line algorithms and in algorithms that use mipmaps. Experiments with software implementations show that the division free algorithms are a factor of two faster, provided that the polygons are not too small. These algorithms are however most profitable when realised in hardware.

CR Categories: I.3.3 [Computer Graphics]: picture/image generation – Display algorithms; I.3.7 [Computer Graphics]: three dimensional graphics and realism – color, shading, shadowing and texture

Keywords: midpoint algorithm; perspective correct; texture mapping; hyperbolic interpolation

1 INTRODUCTION

Well known implementations for perspective correct rendering of planar polygons (such as hyperbolic interpolation[1], also known as rational linear interpolation [5]) require a division per rendered pixel. Such a division is better to be avoided as it is an expensive operation in terms of silicon gates and clock cycles.

Midpoint algorithms [7] are well known for the approximation of conic sections, such as hyperbolae. These algorithms use integer arithmetic only. James Mears [6] points out how the midpoint paradigm may be used for perspective correct texture mapping. As we will point out later, the approach on this web page is not correct. Jan Vondrak mentioned a similar technique, but published his ideas in a usenet posting only.

* bart.barenbrug@philips.com
frans.peters@philips.com
overv@natlab.research.philips.com

Prof. Holstlaan 4, 5656 AA Eindhoven, Netherlands

HWWS 2000 Interlaken Switzerland

Various proposals have been formulated to approximate the hyperbolic curve that is needed by some sort of higher-order polynomial approximation [3][10]. Besides their efficiency, the other main advantage of the midpoint algorithms over these approaches is that no approximations are computed, but exact values (within the precision that is requested).

In this paper we present a family of efficient midpoint algorithms that avoid division operators for perspective correct rendering of plane polygons and that do not require more than a small number of additions per pixel. We present how these midpoint algorithms can be embedded in scan line algorithms and algorithms that use mipmaps. Experiments with software implementations show that the division free algorithms are about a factor two faster, for not too small polygons. These algorithms are however most profitable when realised in hardware.

As such, this family of algorithms is an efficient alternative for those situations where one would otherwise use hyperbolic interpolation.

2 PERSPECTIVE TRANSFORMATIONS

The general form of a perspective transformation is:

$$[x', y', w'] = [u, v, 1] \cdot \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix}$$

where x', y', w' are interpreted as homogeneous 2D screen coordinates and u, v as parameters for the relevant rendering attributes (e.g. texture coordinates). The values $x=x'/w'$ and $y=y'/w'$ are thus given by:

$$x = \frac{Au + Bv + C}{Gu + Hv + I} \qquad y = \frac{Du + Ev + F}{Gu + Hv + I} \tag{1}$$

In (1) x and y are defined as functions of u and v. These formulas may however be rewritten such that u and v are functions of x and y. Then we find expressions of exactly the same form. Indeed, let

$$\begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix}^{-1} = \begin{bmatrix} K & N & R \\ L & P & S \\ M & Q & T \end{bmatrix}$$

then the following holds:

$$[u, v, 1] = [x', y', w'] \cdot \begin{bmatrix} K & N & R \\ L & P & S \\ M & Q & T \end{bmatrix} \quad (2)$$

$$u = \frac{Kx + Ly + M}{Rx + Sy + T} \qquad v = \frac{Nx + Py + Q}{Rx + Sy + T} \quad (3)$$

In the next sections we will consider u and v as functions of x and y. Similar algorithms however may be developed in cases where it is more advantageous to consider x and y as functions of u and v; see e.g. [2].

3 MIDPOINT ALGORITHM

We start with the assumption that all variables to be used are scaled such that they take integer values only. (This is an acceptable assumption as it allows fixed point representations for attributes related to e.g. edge anti-aliasing and bilinear filtering.) We will next formulate the midpoint paradigm in terms of invariants [8].

3.1 Derivation of invariants

The derivation of the invariants is based on relations (2) and (3). Suppose that we have to compute u_i for a series of x_i and y_i values. The midpoint algorithm applied to this problem yields that we have to produce a sequence of u_i values such that:

$$\frac{Kx_i + Ly_i + M}{Rx_i + Sy_i + T} - 0.5 < u_i \le \frac{Kx_i + Ly_i + M}{Rx_i + Sy_i + T} + 0.5 \quad (4)$$

Here u_i is being driven by x_i and y_i. Replacing K, L and M by N, P and Q, and u_i by v_i yields the expressions for v_i as a function of x_i and y_i.

We introduce the following variables:

$$x=x_i \quad y=y_i \quad u=u_i \quad d=2(R\,x+S\,y+T) \quad (5)$$

All d's in the sequence can be assumed to have the same sign. If the matrix in (2) originates from texturing as in OpenGL or Direct3D, $d=0$ corresponds to a projection plane that is coincident with the plane parallel to the screen and through the view point. If we assume that texturing is done after clipping, we know that $d \neq 0$. Furthermore, the pixels of a triangle either all have $d>0$ or all have $d<0$, since they all lie on the same side of the plane parallel to the screen through the viewing point (again because of clipping).

Because of the divisions in (3), we can choose to scale the matrix in (2) by any nonzero factor. We choose to multiply the matrix by *sign(d)* (because of the considerations presented above, a d-value based on any (x,y) pair of the triangle will do). In this way, we can guarantee that $d>0$, which greatly simplifies our derivation and resulting code. Note that this still leaves us the freedom to scale by any positive factor later.

Substituting (5) in the invariant (4), and multiplying by d (using $d>0$) yields:

$$2(Kx+Ly+M) - d/2 < ud \le 2(Kx+Ly+M) + d/2$$

which is equivalent to:

$$ud-(R+2K)x-(S+2L)y-(T+2M)+d > 0 \;\wedge$$
$$ud-(R+2K)x-(S+2L)y-(T+2M) \le 0$$

By introducing $E=ud-(R+2K)x-(S+2L)y-(T+2M)$, we have:

$$0<E+d \;\wedge\; E \le 0 \quad (6)$$

Note that (the value of) d depends on (the values of) x and y; and that E depends on u, d, x and y. During scan-line algorithms, y is increased and x can either be increased or decreased. We will now investigate how the values of E and d can be restored after such changes.

3.2 Restoring the values of *E* and *d*

Incrementing y by 1 requires the following adjustments to our variables:

$d:=d+2S$ and $E:=E+(2u-1)S-2L$.

Incrementing x by 1 leads to:

$d:=d+2R$ and $E:=E+(2u-1)R-2K$.

whereas decrementing x by 1 requires:

$d:=d-2R$ and $E:=E-(2u-1)R+2K$.

To facilitate these adjustments, we introduce variables

$a=(2u-1)R-2K$ and $b=(2u-1)S-2L$.

The changes in value of E (due to changes in x and/or y) can require restoration of invariant (6).

3.3 Restoring invariant (6)

The adjustments needed to restore (6) can be made by incrementing/decrementing u by 1.

Incrementing u by 1 requires:

$E:=E+d;\ a:=a+2R;\ b:=b+2S;$

And decrementing u by 1 requires:

$E:=E-d;\ a:=a-2R;\ b:=b-2S;$

Since $d>0$, E increases when u increases, and E decreases when u decreases. So we know in which direction to adjust u in order to increment/decrement E. However, an increment by 1 in x or y may require multiple unit increments or decrements in u, depending on the value of the partial derivatives $\partial u/\partial x$ and $\partial u/\partial y$, respectively.

We will later introduce mipmapping to bound the values of these jacobians. But still, we cannot simply use a conditional unit increment or decrement of *u*, but we have to use a while loop instead. (This is the point where Mears' algorithm [6] is in error. Mears' algorithm does not have such a while loop, but instead it uses the partial derivatives ∂u/∂x and ∂u/∂y; these are called the gears. If need arises the variables that represent these gears are updated; these updates however are implemented as either an increment or a decrement with 1. This is erroneous: if the hyperbolic curve to be approximated is very steep, then updates with increments or decrements larger than 1 might be required.)

3.4 Algorithm

All of the above gives rise to the following algorithm to compute, e.g. for a sequence of (x_i, y) values, the corresponding u_i values.

```
// Initialisation of x, u, d, a, b and E;
while (x < xmax) do
   x+=1; E+=a; d+=2*R;
   // restore invariants for E:
   while (E  >0) do u-=1; E-=d; a-=2*R od;
   while (E+d≤0) do u+=1; E+=d; a+=2*R od;
od;
```

Note that of the two innermost while-loops, only one can have a guard that is *true*, and it will always be the same during the entire looping over *x*. A case selection therefore could be made in advance.

Note how well the algorithm is qualified for realisation in hardware.

Note moreover, that the algorithm is efficient. Each iteration of both the innermost and the outermost loops requires only three additions and/or subtractions (which might be executed in parallel). The total number of iterations performed by the algorithm is $(u_{max} - u_{min}) + (x_{max} - x_{min})$. Compared with traditional algorithms this saves $(x_{max} - x_{min})$ divisions, at the cost of a more involved setup.

3.5 Setup

The midpoint algorithm does require more setup. In addition to the traditional setup calculations, matrix elements *K–T* must be calculated from (see equation (2)):

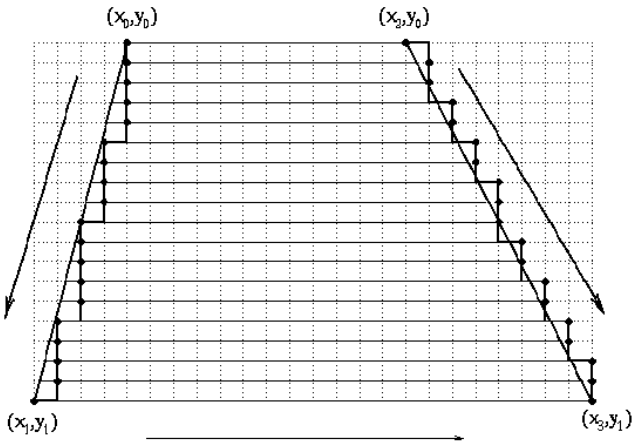
$$\begin{bmatrix} u_0 & v_0 & 1 \\ u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \end{bmatrix} = \begin{bmatrix} x_0' & y_0' & w_0' \\ x_1' & y_1' & w_1' \\ x_2' & y_2' & w_2' \end{bmatrix} \cdot \begin{bmatrix} K & N & R \\ L & P & S \\ M & Q & T \end{bmatrix}$$

Because of the homogenous nature of the equation, we do not have to perform a full inversion of the matrix consisting of the *[x',y',w']* vectors, but can simply calculate the adjoint of this matrix (saving the cost of the division by the determinant). This costs 18 multiplications. The remaining multiplication with the matrix consisting of the *[u,v,1]* vectors also costs 18 multiplications. In Section 4.4 we will show the effects on overall computation time.

4 INCORPORATION IN A SCANLINE ALGORITHM

4.1 Extending a scanline algorithm

Classical scanline algorithms traverse triangles that need to be rendered by having an inner loop that traverses in the screen-*x* direction, which is embedded in an outer loop which traverses the screen-*y* direction, over the range of the triangle. During this traversal, the algorithm maintains two active edges between which the scanlines are drawn. We will call these edges the left and the right edge (`l` and `r` for short). Each of these edges has to be discretised in order to provide the start and end points of a scanline. This is shown in the figure below.



In [8], a derivation of a midpoint algorithm is presented which updates the *x*-values of the points on the active edges as the *y*-coordinate is increased. For this algorithm (using the end points (x_0,y_0) and (x_1,y_1) of the left edge), the variables *dx* and *dy* are introduced, as well as invariants:

$$dx = x_1 - x_0 \quad \wedge \quad dy = y_1 - y_0 \quad \wedge$$

$$x_0 + \frac{dx}{dy}(y - y_0) \le x < x_0 + \frac{dx}{dy}(y - y_0) + 1$$

Introduction of variable *e*, defined by $e=dx(y-y_0)-(x-x_0)dy$ leads to the invariant:

$$e \le 0 \ \wedge \ 0 < e + dy$$

This active edge traversal is performed in the following code:

```
y:=y0; dy:=y1-y0;
xl:=x0; dxl:=xl-x0; exl:=0;
xr:=x2; dxr:=x3-x2; exr:=0;
// now traverse the scanlines:
while (y<y1) do
   // restore invariants for xl and xr:
   while (exl+dy<=0) do xl-=1; exl+=dy od;
   while (exl   > 0) do xl+=1; exl-=dy od;
   while (exr+dy<=0) do xr-=1; exr+=dy od;
   while (exr   > 0) do xr+=1; exr-=dy od;
   // now we are ready to render a scanline:
```

```
    Scanline(xl,xr,y);
    // increase y to go to the next scanline:
    y+=1; exr+=dxr; exl+=dxl
od;
```

The `Scanline` procedure draws a horizontal line at height `y` from `xl` to `xr`. We will start with this `Scanline` procedure to add the midpoint algorithm for the texture coordinates. The procedure has a loop (we will call this the "inner loop" from now on) in which `x` is incremented from `xl` to `xr`, in order to traverse the scanline. For each of the pixels that are encountered, we will need to update the *u* and *v* coordinates to be able to retrieve the correct colour from the texture map.

We will assume that for each of the end points the texture coordinates are given (so the end point with *x*-coordinate x_i has texture coordinate (u_i,v_i).

The following pseudo code shows the enhanced `Scanline` procedure, using the midpoint algorithm to keep up-to-date values for *u* and *v*. The classical scanline code is in italics, while the additional midpoint code is in regular font. Any interpolation of *z*-values for the benefit of *z*-buffering is ignored here.

```
Scanline(xmin,xmax,y,umin,vmin: integer;
         eu,ev,au,av,d: real)
{
   // starting values for E,a and d
   // for u and v are calculated
   // in the outer loop
   x:=xmin; u:=umin; v:=vmin;
   while (x<xmax) do
     // restore invariants for u and v:
     while (eu  >0) do u-=1; eu-=d; au-=2*R od;
     while (eu+d≤0) do u+=1; eu+=d; au+=2*R od;
     while (ev  >0) do v-=1; ev-=d; av-=2*R od;
     while (ev+d≤0) do v+=1; ev+=d; av+=2*R od;
     // render the pixel:
     pixel[x,y]:=texel_colour(u,v);
     // go to the next pixel:
     x+=1; eu+=au; ev+=av; d+=2*R
   od;
}
```

As noted in Section 3.4, only one of the two while-loops that update *u* (the same for *v*) can have a guard that is *true*, and it will always be the same during the entire looping over *x*. A case selection could be made in advance, yielding four cases for the example above (two possibilities of each of the two loop-pairs). Many modern renderers however need to support multi-texturing; e.g. Direct3D allows up to eight *(u,v)* pairs, leading to 2^{16} different cases (2^{18} in the outer loop where also two *x*-values have to be interpolated). Since the presence of both loops only means a slight performance degradation (since one more guard has to be checked), and these loops otherwise do not interfere with each other, no case analysis is made. In hardware, both guards may be evaluated in parallel, if so required.

The outer loop is a bit more complex, since both *x* and *y* are adjusted there. The following code shows the additions for maintaining *u* and *v* values for the left edge:

```
// initialisation:
y:=y0; dy:=y1-y0;
xl:=x0; dxl:=x1-x0; exl:=0;
xr:=x2; dxr:=x3-x2; exr:=0;
d:=2*(R*xl*S*y+T);
u:=u0; au:=(2*u-1)*R-2*K; bu:=(2*u-1)*S-2*L;
v:=v0; av:=(2*v-1)*R-2*N; bv:=(2*v-1)*S-2*P;
eu:=u*d-((R+2*K)*xl+(S+2*L)*y+T+2*M);
ev:=v*d-((R+2*N)*xl+(S+2*P)*y+T+2*Q);
// active edge traversal:
while (y<y1) do
   // restore invariants for xl and xr:
   while (exl+dy<=0) do
    xl-=1; exl+=dy; eu-=au; ev-=av; d-=2*R
   od;
   while (exl   > 0) do
     xl+=1; exl-=dy; eu+=au; ev+=av; d+=2*R
   od;
   while (exr+dy<=0) do xr-=1; exr+=dy od;
   while (exr   > 0) do xr+=1; exr-=dy od;
   // restore invariants for u and v:
   while (eu +d <=0) do
     u +=1; eu +=d; au+=2*R; bu+=2*S
   od;
   while (eu    > 0) do
     u -=1; eu -=d; au-=2*R; bu-=2*S
   od;
   while (ev +d <=0) do
     v +=1; ev +=d; av+=2*R; bv+=2*S
   od;
   while (ev    > 0) do
     v -=1; ev -=d; av-=2*R; bv-=2*S
   od;
   // now we are ready to render a scanline:
   Scanline(xl,xr,y,u,v,eu,ev,au,av,d);
   // increase y to go to the next scanline:
   y+=1; exr+=dxr; exl+=dxl;
   eu+=bu; ev+=bv; d+=2*S
od;
```

Since expressions *2R* and *2S* are used so often, a variable can be introduced for each of these to speed up the calculations.

4.2 Use of mipmaps

The algorithm as described in the previous section just computes texture coordinates, and more or less implements point sampling. We now show how to use mipmaps (more on mipmaps can be found in [4]). Mipmaps provide better filtering than the point sampling, and also limit the number of iterations required in the inner loop to correct the invariants for *u* and *v*: as proper mipmap level selection ensures that the values of jacobians like $\partial u/\partial x$ never exceed a value of two, at most two iterations for the invariant correction are required.

The following assumes a form of mipmap organisation where we have smaller versions of the original texture map, each scaled down in both *u* and *v* directions by a factor two with respect to the level below (so mipmap level 0 is the original texture map, level 1 is the original scaled down by a factor 2 in both *u* and *v*, level 2 is the original scaled down by a factor 4 in *u* and *v*, etc.).

4.2.1 Mipmap level coordinate systems

For each of the mipmap levels, there is a different correspondence between the texture coordinates in that mipmap level and the screen coordinates. We can choose the coordinate system for our higher mipmap levels such that for mipmap level *m*:

$$u = \frac{1}{2^m}\frac{Kx + Ly + M}{Rx + Sy + T}$$

(and similarly for *v*). For the variables we introduced, we can account for this by multiplying *R*, *S* and *T* by 2^p, and dividing *K*, *L*, and *M* by 2^q, where $p+q=m$. The values *p* and *q* can be chosen in order to make best usage of the available word length for the variables K, L, …T in fixed point representation If for instance we set $q=m$, $p=0$ we get:

$d=2(Rx+Sy+T)$
$E=ud-(R+(2/2^m)K)x-(S+(2/2^m)L)y-(T+(2/2^m)M)$
$a=(2u-1)R-(2/2^m)K$
$b=(2u-1)S-(2/2^m)L$

4.2.2 Mipmap level switching

Now that we know how to adjust our variables in correspondence to the current mipmap level, we have to determine how to decide on which mipmap level to use. (A wrong level either leads to aliasing or to blurring.) Ideally, there is an approximate 1:1 correspondence between the distance between adjacent texels and the distance between adjacent pixels. That is: we want to keep the absolute values of ∂u/∂x and ∂u/∂y close to one (and similar for *v*). An often used formula (where *log* stands for ^2log) is:

$$m = \log\left(\max\left(\left\|\left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right)\right\|, \left\|\left(\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}\right)\right\|\right)\right) \qquad (7)$$

In order to determine the mipmap level, we need to determine expressions for the derivatives. In terms of our variables (using the *u* and *v* of the current mipmap level), the following expressions can be derived (for *u*, and similar for *v*):

$$\frac{\partial u}{\partial x} = -\frac{a+R}{d} \qquad \frac{\partial u}{\partial y} = -\frac{b+S}{d}$$

We can approximate these with:

$$\frac{\partial u}{\partial x} = -\frac{a}{d} \qquad \frac{\partial u}{\partial y} = -\frac{b}{d}$$

Since *a* changes with *2R* per texel, and *b* with *2S*, this approximation only means that the mipmap level is switched half a texel earlier or later.

These expressions are based on the values of *a* and *b* for the current mipmap level. Hence the resulting value for *m* in (7) is relative to the current mipmap level and therefore a mipmap level change requires adjusting *a* and *b*. Rewriting (7) using Euclidean distance measure yields:

$$\Delta m = \frac{1}{2}\max\left(\log\left(a_u^2 + b_u^2\right), \log\left(a_v^2 + b_v^2\right)\right) - \log(d)$$

and using Manhattan distance:

$$\Delta m = \max\left(\log\left(|a_u| + |b_u|\right), \log\left(|a_v| + |b_v|\right)\right) - \log(d)$$

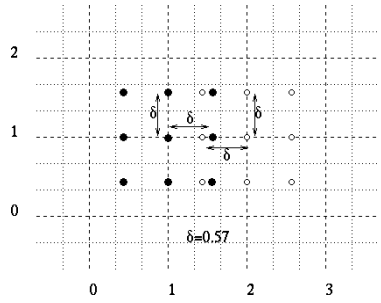
Since we are only interested in integer values for Δm, we can take the integer *log*. If the real numbers *a*, *b* and *d* are represented as floating point numbers, taking the integer *log* is simply a matter of selecting the exponent. For a fixed point representation, this *log* is easily evaluated by determining the position of the most significant "1" bit. So these expressions are easily evaluated (note that taking the logarithm also removes the need to explicitly perform the division of *a* or *b* by *d*).

For each pixel, we can evaluate Δm and see if we need to switch mipmap level(s) by adjusting/recalculating the values for *a*, *b* and *E*. Since the precise location of a mipmap level switch is usually not critical, we can even afford to perform this test, say, only once every 2 or 3 pixels.

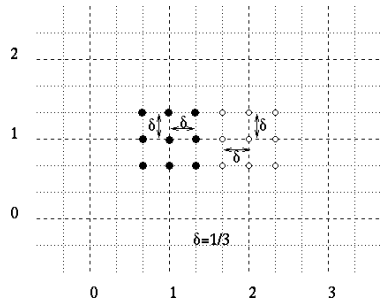
4.3 Super-sampling

Many anti-aliasing methods exist to improve image quality. We need a method which provides full anti-aliasing (and not just edge anti-aliasing) to handle texture features. Super-sampling is a technique that combines well with the midpoint algorithm because the midpoint algorithm is well suited for triangles with many pixels (see Section 4.4), and super-sampling enlarges the number of samples to be processed in a triangle.

Two versions have been implemented. The first version 3x3 super-samples the neighbourhood of a pixel, parameterised by a distance δ that tells how far the super-samples are apart from each other. This is depicted in the picture below in which the super-samples for pixels (1,1) are depicted in black, and those for (2,1) as an open circle. The samples are obtained by rendering a triangle nine times into an accumulation buffer, each time shifted by δ with respect to each other. The parameter δ allows balancing the amount of blurring versus aliasing. Visual inspection of test examples revealed that δ=1/3 is a (near) optimum case.



The case of δ=1/3 is a special case, because (as seen in the picture below), the super-samples are now on a regular grid. This allows for an optimisation: instead of rendering a triangle nine times, it is simply rendered once at three times (in both x and y direction) the resolution, thereby avoiding a lot of setup, and using more spatial coherence.



This was also implemented (for the general case of *n*x*n* super-sampling with the distance between super-samples being 1/*n*). Further optimisation is then possible by only checking for mipmap switches once every *n* samples, in such a way that the check (and possible switch) happens on a "middle super-sample", so that the combined colour results from the blending of super-samples from different mipmap levels. This helps hide any artefacts associated with mipmap level switching. For now, the smoothing filter kernel was a block, but a more advanced filter profile could be applied in the future.

The two pictures below show a texture map, textured on a rotated square without super-sampling (Figure 1a), and that map textured with 3x3 super-sampling (Figure 1b).

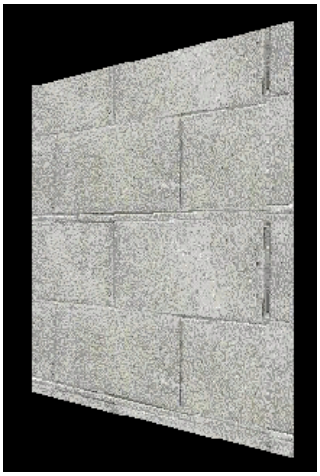


Figure 1a

Figure 1b

It is clear that Figure 1a shows many problems: there are jaggies (both at the edges of the textured polygons and internally due to sharp contrasts in the texture map), and detail that is a bit further away (like the left-most vertical separation between the grey bricks) disappears in the aliasing noise that results from the fine detail in the original texture map (Figure 2). These problems are largely alleviated in Figure 1b.

Figure 2

Figure 3

4.4 Test results

The scanline algorithm with the midpoint-based texture mapping algorithm was implemented on a PC. For comparison, we also implemented a version of the same scanline algorithm using the perspective division per pixel for texturing. Mipmapping, as well as the two variants of super-sampling mentioned in Section 4.3, were implemented for both midpoint and division algorithms.

The scan converter (which renders a square consisting of two triangles, based on a 256x256 texture map) was tested with three different animation sequences:

1. A textured square which rotates such that the projected area varies between 0 and 100%. This test demonstrates the effect of large differences in the *u* and *v* derivatives. Because of the rotation, the aliasing of any edges (of the square and within the texture) can be studied. Examples of frames from such a test animation are the pictures in Figure 1a and 1b.

2. A square receding along the z-axis. This one shows the effects of the mipmapping, as the mipmap level is switched while the square gets farther away. An example of a frame of such a test animation is shown in Figure 4a below (Figure 3 is the original texture map).

3. A square shrinking in size, and only showing the corresponding part of the texture. This test allows varying the triangle size while staying at the same mipmap level. An example of a frame from such a test animation is shown in Figure 4b below.

Figure 4a

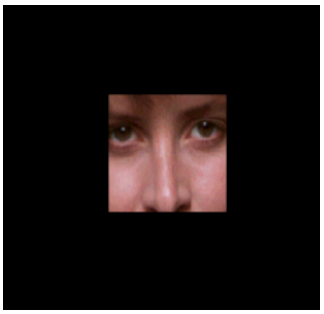


Figure 4b

Performance measurements were taken for all three tests, measuring the rendering time per frame as a function of the number of pixels that are rendered during that frame. The results obtained show that the midpoint implementation is typically twice as fast as the implementation with the divisions, for large enough triangles. The break-even point (which varies greatly between different tests) was always below 10 pixels per triangle (and for such small triangles it is questionable if you want perspective correct mapping).

Of course the relative performance differences as measured on a PC implementation are only indicative of the performance of a hardware implementation. However, some observations can be made:

- The `while ("e compares favourably to 0")` loops for re-enforcing the invariants for the different variables can be executed in parallel. With a division-based solution this costs a lot of silicon, but since these loops now only contain additions and subtractions, they can be implemented with very little silicon. This is very advantageous for the support of fast multi-pass algorithms where many variables need to be adjusted in this way. These passes can all be done in parallel at low hardware cost. The only concern here is the hardware required for the switch in mipmap levels. But since mipmap level switching can be delayed if necessary, such hardware can be shared along the passes.

- The exponent-selection to perform the integer *log* can be hardwired in hardware (while it needs some shifting and masking in software), so the mipmap level switch detection can be implemented faster in hardware. The mipmap level switch detection might also be performed in parallel: the exact mipmap level switch position is not critical, so the detection could be done in parallel, only interrupting the "normal execution" when it is detected that the mipmap level needs to be switched. The associated small latency is probably not a problem.
- Our implementation so far still uses floating point numbers for matrix elements *K–T*, and corresponding variables E, *d*, *a* and *b*. Using a fixed point representation for these should also further speed up the calculations.

5 FUTURE WORK

The promising results presented in the previous section indicate that further studies in this area are warranted.

Despite all extensions, the algorithms described still perform point sampling. Appropriate filtering should be added to better deal with aliasing effects.

In case an update in the outermost loop of the algorithm as presented in Section 3 leads to a large number of updates in the innermost loop (which could be the case when using a fixed point representation for *u* and *v*, instead of the integer representation presented in this paper; or when not using mipmapping), further speedups are possible. Let i_x be the number of iterations of the innermost while-loop after the most recent update of *x*, then i_x, i_{x+1}, i_{x+2}, … is either a non-increasing or a non-decreasing sequence of values. This may be exploited by the introduction of a variable *i* to record this number of iterations. Instead of incrementing or decrementing *u* a number of times with 1, *u* might be updated with *i* followed by some updates with 1 (while adjusting *i*).

The *i*-values for the different variables are a discrete approximation of the jacobians for those variables (corresponding more or less to the *gears* in [6]), and can as such be used to perform mipmap level selection in an even more straightforward manner than the method described in Section 4.2.2. Only the upper few bits of the *i*-values are required for this purpose. The jacobians can also be used to determine (an approximation of) the footprint of a pixel, which is useful when performing filtering.

In the current implementation, the real-valued variables *K–T* and *E*, d, *a* and *b*, are maintained in floating point representation. A fixed point implementation is faster, and cheaper to build in hardware. To this end, some attention should be directed to scaling the transformation matrix such that it its values fit within the fixed point range. Because of the homogeneous nature, this is not a problem, apart from determining the correct scale factor.

6 CONCLUSIONS

We have derived elegant, perspective correct rendering algorithms that are free from divisions in their loops. The tests in the Section 4.4 show that for scanline based rendering, the midpoint based algorithm achieves about twice the performance compared to the currently used algorithms, when implemented on a PC. Hardware implementation is expected to reach similar performance gain, while less hardware is required (no division hardware is needed). Since less hardware is required for the inner loops, parallel texturing units to implement multi-texturing become viable, providing an additional performance gain. The extra setup costs only matter for triangles which are so small that perspective correct rendering is not important.

References

[1] Blinn, J.F. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications*, pp. 89–94, July 1992.

[2] Catmull, E., A.R. Smith. 3-D Transformations of Images in Scanline Order. *Computer Graphics (SIGGRAPH '80 Proceedings)*, vol. 14, no.3, pp. 279–285, July 1980.

[3] Demirer, M. and R.L. Grimsdale. Approximation Techniques for High Performance Texture Mapping. *Comput.&Graphics*, vol. 20, no. 4, pp. 483–490, 1996

[4] Ewins, J.P., Waller, M.D., White, M., Lister, P.F. MIP-map level selection for texture mapping, in *IEEE Transactions on Visualization and Computer Graphics, vol. 4, no 4.*, pp. 17–29, Oct.–Dec. 1998

[5] Heckbert, P.S. and H.P. Moreton. Interpolation for polygon texture mapping and shading, in *State of the Art in Computer Graphics*, D. Rogers and R. Earnshaw, editors; New York: Springer-Verlag, 1991.

[6] Mears, J. The Midpoint Algorithm for High Speed Graphics. December, 1996, Web page at URL: http://www.fitzharrys.freeserve.co.uk/midpoint/index.htm

[7] Pitteway, M.L.V. Algorithms for drawing ellipses or hyperbolae with a digital plotter. *Computer Journal*, 1967, **10**(3), pp. 282–289.

[8] Van Overveld, C.W.A.M., Course Notes Computer Graphics (2K900), 1988. Eindhoven University of Technology.

[9] Van Overveld, C.W.A.M, Applications of the method of invariants in computer graphics, in *Data Structures for Raster Graphics*, L.R.A. Kessener, F.J. Peters and M.L.P. van Lierop editors; *Eurographic Seminars*, Berlin etc. Springer-Verlag, 1986.

[10] Wolberg, G. *Digital Image Warping*. IEEE Computer Society Press, 1990