Yet Another Anti-Aliasing Technique for Computer-Generated Images

Francisco Morán

Grupo de Tratamiento de Imágenes
E.T.S. Ing. Telecomunicación (C-306)
Universidad Politécnica de Madrid
28040 Madrid - SPAIN
Tel.: +34.1.336.7353
Fax: +34.1.336.7350
E-mail: fmb@gti.ssr.upm.es
WWW: http://www.gti.ssr.upm.es./~fmb/

Abstract: We describe an efficient variation on the z-buffer algorithm for hidden-surface removal in computer-generated images, which allows to selectively post-filter such images in order to reduce the artifacts due to sub-sampling. With a moderate increase in both the computational cost and the memory needs of the classic z-buffer, we filter only those areas where aliasing is most noticeable and improve considerably the subjective resolution of the resulting images. This technique has been added successfully to the "flying logos" software package developed for a commercial character generator used in TV post-production.

0. Introduction

Aliasing occurs due to point-sampling an original image containing frequencies higher than the sampling one, and can be fought by either increasing the sampling frequency or pre-filtering the original image to limit its bandwidth. Unfortunately, the first solution implies increasing the number of pixels of the resulting image, which is limited by the physical constraints of the current hardware. And the second demands that the original image be area-sampled, which in turn means to obtain several working samples to compute each final pixel by weighted or unweighted averaging. This super-sampling process, be it regular or stochastic, increases always drastically the CPU time required by any image processing application. In the particular case of 3D scenes rendering, as the "original image" does not really exist, super-sampling it means in fact performing several times by pixel all the calculations needed for hidden-surface removal, lighting and shading. It means as well storing those intermediate results, so when this is done on a 3x3 or 4x4 samples per pixel basis, not only the processing time, but also the amount of memory needed to render the scene, can be easily multiplied by a factor of ten or more.

Of the different variations on the z-buffer designed to reduce aliasing, the a-buffer presented in 1984 by L. Carpenter is probably the more elegant and general, and it requires only a small increase in cost, in terms of memory. Nevertheless, the idea behind the a-buffer remains to perform a super-sampling of the "original image", and thus its computational cost remains noticeably higher than that of the standard z-buffer, in spite of its brilliant optimizations.

1. Strategy and Basic Algorithm

It is clear that, as pixels do have a non-null area, some pixels are completely interior to a polygon, but some others are only partially covered by it. Any reasonable span filling algorithm will label as “interior to the polygon” the pixels that are at least half-covered by the polygon, and will assign the same colour (supposing, of course, that flat shading is used) to all such pixels.

The problem is that the pixels lying on the boundaries of the polygon are also partly covered by other polygons, so it would be wiser to have their colour be some blend of the colours of the polygons that share them, as shown in figure 1. In fact, all polygons sharing a pixel P should contribute to P’s colour, and they should do it proportionally to the area they cover inside P.

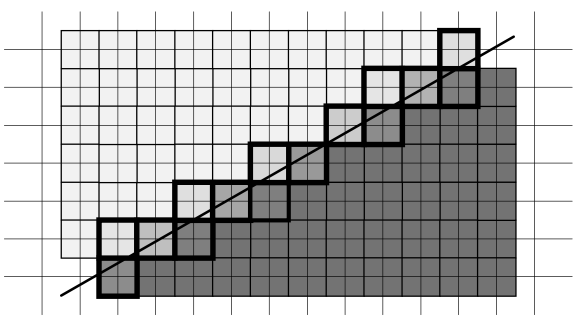

Figure 1: Colour blending along edges.

The proportional contribution of each of a boundary pixel’s neighbours to the final colour of that pixel, which is precisely the goal behind weighted area-sampling, is usually achieved by rendering the scene at many working samples per final pixel. Our technique lets the z-buffer algorithm work at a minimum resolution of only one sample per pixel, and therefore generate artifacts, but keeps track of where the artifacts will appear, in order to selectively filter the aliased image afterwards.

The basic algorithm could be expressed as follows:

1a. project all the polygons in the scene “z-buffer-wise”, AND

1b. keep track of their edges’ position in both the frame and depth buffers;

2. once the spans of the last polygon have been filled, walk along each edge previously stored, and replace each visible pixel’s colour with a blend of its neighbours’ colours.

It is important to note that it is impossible to know what polygons share a pixel (i.e., what colours have to be blended to obtain that pixel’s final colour) before all polygons have been rendered. Furthermore, it is also impossible, before all polygons have been rendered, to know whether an edge will or will not be visible, which is of crucial importance, because this colour blending process must not take place along hidden edges. But once the rendering step has finished, the blending step can be efficiently performed, thanks to the following:

- edge traversals are made by an incremental algorithm that follows always 4-connected lines (except when the mathematical line enters or leaves a pixel precisely by its corner, as shown in figure 1) and that performs at the same time, also incrementally, the partial area coverage calculations;
- only two neighbours of a pixel contribute to the blend: for “mostly horizontal” edges (i.e., those with $|slope| < 45º$), only the North and South neighbours of each pixel are considered; and for “mostly vertical” edges, only its West and East neighbours contribute to the blend. We tested several other more expensive combinations, but found the results of this simpler one satisfactory enough.

2. Detailed Algorithm

2.1. *Z*-coordinate trim at boundary pixels

At the core of any implementation of the z-buffer algorithm, there is a span filling step which has to perform (usually incremental) calculations to store in the depth-buffer the *z*-coordinate of each 3D point of the polygon being projected, at the same time as the corresponding colour is stored as a pixel in the frame-buffer. The Mother of All Problems mentioned before —a pixel's not being a mathematical point with a null area— strikes back during that process, as it is erroneous to assume that all the 3D points from a polygon corresponding to a single pixel share the same *z*-coordinate.

With boundary pixels, this error is even worse, because they are partially covered by more than one polygon. The result is that, for a pixel lying on the edge shared between polygons P1 and P2 (cf. figure 2), the span filling process will store in the depth-buffer a *z*-coordinate satisfying «$a_1x+b_1y+c_1z = d_1$» if the pixel is mostly covered by P1, and «$a_2x+b_2y+c_2z' = d_2$» if it is mostly inside P2.

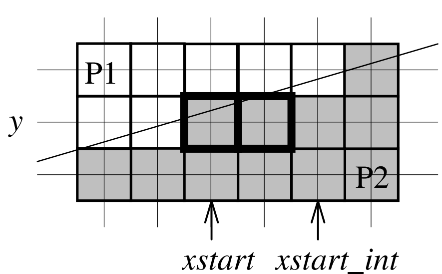

Figure 2: Boundary and interior pixels.

But once a polygon's spans have been filled, all data concerning it are discarded, and we wanted to store only the device coordinates (row, column and depth) of its edges. So, when an edge traversal would be performed during the filtering step, the plane equations of the different polygons sharing it would not be available. That is why we decided to assign to the boundary pixels of each polygon the exact same *z*-coordinate that would be reconstructed, during the filtering step, by linear interpolation between those of the corresponding edge ends. In figure 2, those pixels have been shown for P2: during the filling of its spans in row *y*, pixels verifying «$xstart \leq x < xstart\_int$» would be affected by this "*z*-coordinate trimming mechanism".

2.2. Edge *priority* in the blending process

To avoid cumulative effects in scenes with many edges, we decided to mark the pixels affected by the blending process, along with their neighbours, as "unreblendable". The introduction of this mechanism showed more clearly the already existing dependence of the result on the order in which the edges were processed.

To eliminate it, we decided to assign a *priority* to each edge, so that *primary* edges would be processed before *secondary* ones during the filtering step. The reason behind this concept is that two polygons with similar normal vectors (cf. figure 3) will most likely have similar colours when lit and shaded, so the edge they share is of low priority, as opposed to an edge shared by two polygons with quasi-opposite (or even quasi-perpendicular) normal vectors, which is essential to our technique, as it will probably generate a very noticeable "jaggie".

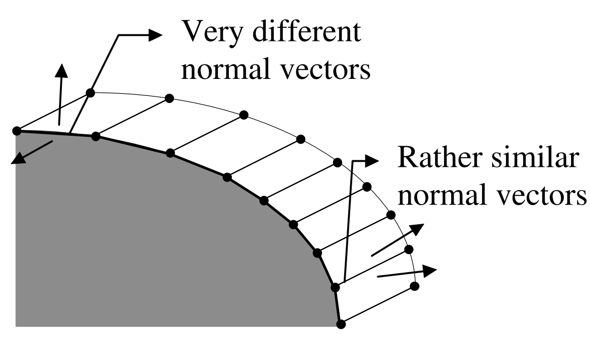

Figure 3: Primary and secondary edges.

2.3. Summary

The complete algorithm, including the improvements over the original one, is this:

1a. project all the polygons in the scene according to the z-buffer algorithm, BUT "trimming" the *z*-coordinate of the pixels in the spans' ends, AND

1b. keep track of their edges' device coordinates on an ordered list (sorted by decreasing priority);

2. once the spans of the last polygon have been filled, walk along each edge previously stored (processing first high-priority edges, then low-priority ones), and replace each visible pixel's colour with a blend of its neighbours' colours and mark both the pixel and its neighbours to avoid cumulative effects.

3. Results and conclusions

The images shown in figures 4 and 5 have been generated with the MS-Windows3.1 version of "3D Text", and correspond to a very simple geometry of only 71 polygons rendered with Phong shading, for which the anti-aliasing step took only an extra 4%. For more complex models, where the number of boundary pixels is comparable to that of completely interior pixels, the extra CPU time is of course higher, but always much smaller than the one already needed by the rendering process itself. This can be seen in table 1, which shows the extra CPU time required by the anti-aliasing step of the MS-Windows95 version of "3D Text".

This technique is efficient and gives good-quality results for scenes with a few hundred big- or medium-sized polygons, which are typical in a "flying logos" application as the one this algorithm was designed for. However, it is not as general as other anti-aliasing techniques that perform a more rigorous, but also more expensive, weighted area-sampling across the whole image. In particular, it does not handle the aliasing due to texture-mapping: our main concern was to improve as much as possible the subjective resolution of the rendered images by eliminating the jagged edges from the polygons' borders.

Polygons	Flat	Gouraud	Phong
54	+11%	+10%	+2%
90	+13%	+12%	+2%
180	+14%	+13%	+3%
360	+14%	+13%	+4%
630	+23%	+18%	+5%
1260	+27%	+24%	+9%

Table 1: CPU time needed by filtering step.

4. References

1. L. Carpenter: *The A-buffer, an Antialiased Hidden Surface Method*, ACM Computer Graphics, 18-3 (proc. of SIGGRAPH'84), 1984, pp. 103-108.
2. F. Crow: *The Aliasing Problem in Computer Generated Shaded Images*, Communications of the ACM, **20**-11, 1977, pp. 799-805.
3. F. Crow: *A Comparison of Antialiasing Techniques*, IEEE Computer Graphics and Applications, **1**-1, 1981, pp. 40-48.
4. J. Foley, *et al.*: *Computer Graphics, Principles and Practice* (2nd. ed.), Addison-Wesley, 1990, pp. 617-646, 693-695.
5. M. Pitteway, D. Watkinson: *Bresenham's Algorithm with Grey Scale*, Communications of the ACM, **23**-11, 1980, pp. 625-626.
6. G. Wolberg: *Digital Image Warping*, IEEE Computer Society Press, 1990, pp. 8-9, 106-111, 163-186.

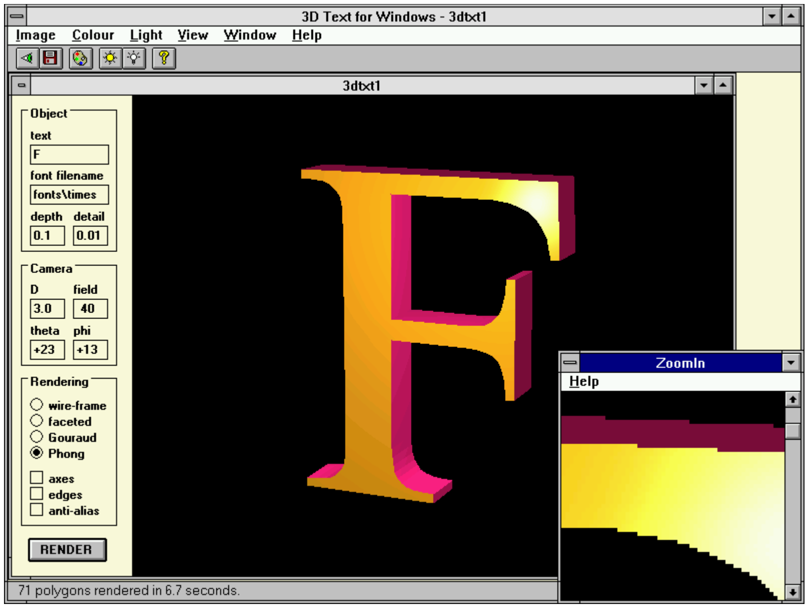

Figure 4: Aliased “F”.

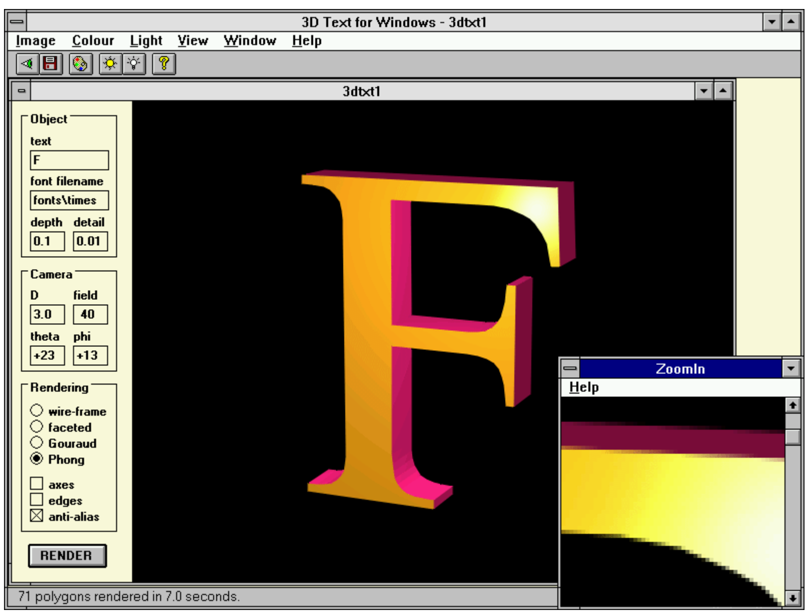

Figure 5: Anti-aliased “F”.