A Fast Anti-Aliasing Technique for Computer-Generated Images

Francisco Morán and Narciso García

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.

Keywords: Computer-generated images, z-buffer, jagged edges, anti-aliasing, adaptive filtering.

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 [5, 7, 9]. 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 obtaining 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.

Several algorithms can be used inside a 3D rendering program to solve the hidden-surface removal problem and correspondingly, different anti-aliasing techniques have been designed [3, 4]. Among such algorithms, the most commonly used is probably E. Catmull's z-buffer, which was improved by its author himself to reduce aliasing, only at the expense of efficiency [2]. Of the other variations on the z-buffer designed to reduce aliasing, L. Carpenter's a-buffer [1] 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 whole "original image", and thus its computational cost remains noticeably higher than that of the standard z-buffer, in spite of its brilliant optimizations.

The anti-aliasing technique presented below was originally designed for the "flying logos" software package developed for a commercial character generator used in TV post-production, which was built on top of an M68020 processor with no specific 3D graphics hardware. Efficiency being the major design concern, an idea was sought to selectively filter only those areas of the image where the artifacts due to aliasing are most striking to the eye, instead of indiscriminately filtering the whole image. In fact, as either Gouraud's or Phong's smooth shading would be applied to the interior of the polygons and texture mapping was not one of the required features of the renderer, abrupt colour transitions were to be found only on the edges of the polygons: the goal was then to eliminate the infamous "jaggies". But the location of those edges would be known beforehand, so it would be possible to perform that selective filtering of the image after having let a slightly modified z-buffer generate the aliasing artifacts inherent to its original version.

1. Strategy and Basic Algorithm

At the core of any implementation of the z-buffer algorithm, there is a span filling routine [5] 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. It is clear that, as pixels do unfortunately have a non-null area, some of them 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.

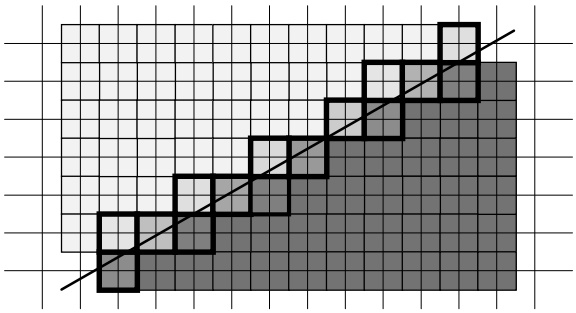

Figure 1: Colour blending along edges.

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 (portions of otherwise visible) 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: the North and South ones for "mostly horizontal" edges (those with «abs (slope) < 45»), and the West and East ones for "mostly vertical" edges. 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

The Mother of All Problems mentioned before —a pixel's not being a mathematical point with a null area— strikes back during the span filling process to affect also the *z*-coordinates stored in the depth buffer, 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, but «$a_2x+b_2y+c_2z' = d_2$» if it is mostly inside P2.

The problem is that, in the standard z-buffer algorithm, all data concerning a polygon are discarded once its spans have been filled: it is impossible to know, after the rendering step is done, what polygons share a particular edge — let alone the plane equations of those polygons... Nevertheless, as we would store the device coordinates (row, column and depth) of the edge anyway, we decided not to save extra information regarding the plane equations of the polygons sharing it, but to assign instead, to the boundary pixels of each polygon, the exact same *z*-coordinate that would be reconstructed, during the filtering step, by linear interpolation between the *z*-coordinates of the corresponding edge ends.

In figure 2, the left boundary pixels of P2 in row *y* have been highlighted. They are those whose column *x* verifies «*xstart* ≤ *x* < *xstart_int*», and are the only ones that will be affected by our *z*-coordinate trimming mechanism: the ones satisfying «*xstart_int* ≤ *x*» will be assigned a *z*-coordinate verifying P2's polygon equation, according to the standard z-buffer mechanism.

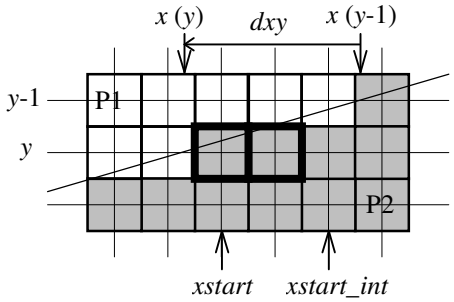

Figure 2: Boundary and interior pixels.

The calculation of *xstart* and *xstart_int*, which are the integer abscissas of the pixels on and to the right of which one is respectively mostly and completely inside the polygon, is easy. Letting $x = x(y)$ be the exact abscissa of the crossing of the considered edge with row *y* and $dxy = x(y) - x(y-1)$, and furthermore letting "ceil (*x*)"denote the smallest integer greater than *x* and "round (*x*)" the integer closest to *x*, *xstart* and *xstart_int* can be expressed as follows:

- *xstart* = ceil (*x*)
- *xstart_int* = round (*x* + abs (*dxy*) / 2) + 1

Of course, a similar reasoning leads to analogous results regarding the abscissas of the pixels on and to the left of which one is completely and mostly inside the polygon. Letting "floor (*x*)" denote the largest integer smaller than *x*, those abscissas are respectively:

- *xstop_int* = round (*x* − abs (*dxy*) / 2) − 1
- *xstop* = floor (*x*)

2.2. Marking of already blended pixels

In all synthetic images, there are areas with many more edges than others: for instance, the side of a 3D character created by extruding a curved, 2D character, will have many more polygons than its front and back faces, and that will create a very heterogeneous distribution of the edge density in the corresponding image. To avoid cumulative effects in "edge-rich" areas, we decided to mark the pixels affected by the blending process, along with their neighbours, as "unreblendable", by assigning to them a null *z*-coordinate. At the cost of some more memory, a bitmap of the same dimensions than the frame and depth buffers could be created to implement this marking mechanism, not to distort the original meaning of the *z*-coordinates stored in the z-buffer.

The introduction of the marking of already blended pixels and of their neighbours showed more clearly the already existing dependence of the result of the filtering step on the order in which the edges were processed, and thus the necessity of introducing the concept of "edge priority", that enables the filtering step to produce deterministic results.

2.3. Edge priority in the blending process

The problem mentioned above affects pixels belonging to several edges, as the ones shown in figure 3: in the case of pixel "a", the neighbours considered for the colour blending process would be different depending on which edge is processed first, as one is "mostly horizontal" and the other "mostly vertical"; in the case of pixel "b", the same neighbours would be considered, but their colours would be blended according to different proportions.

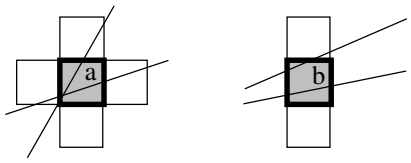

Figure 3. Pixels belonging to several edges.

To solve this problem, 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 4) 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, that is essential to our technique, as it will probably generate a very noticeable "jaggie".

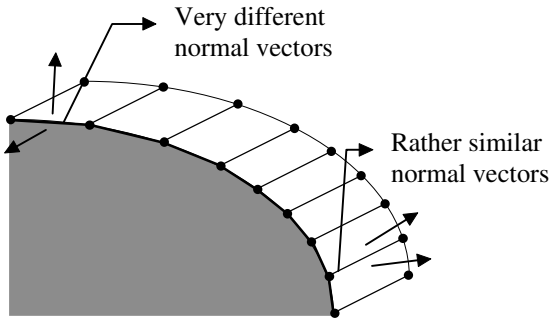

Figure 4: Primary and secondary edges.

As to the implementation of this mechanism, a classification of an edge based on a single bit would probably have been sufficient for objects like 3D characters, but we decided to code their priority with a whole byte, according to the angle formed by the two polygons sharing the edge. That value

must be assigned by the modeler, so the rendering module may sort the edges on the ordered list that is created during the span filling step, and process them afterwards in the correct order during the filtering one.

2.4. 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 edge priority (which must be specified by the modeler);

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, unmarked 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 5 and 6 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's 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, specially if Phong's shading is used. 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 the program derived from "3D Text", which handles now arbitrary geometries (it is not limited to extruded text).

Number of polygons	Shading		
	Flat	Gouraud's	Phong's
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.

Our variation on the z-buffer algorithm, designed to reduce aliasing in computer-generated images, 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 conceived 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.

4. Acknowledgements

The mechanisms introduced to improve the original idea are partly due to José Javier García, whose help in the programming tasks and constant encouragements —"¡esta vez sí que ya está!"— were of the greatest importance. I would like to thank also José María Martínez for having been patient enough to review the different versions of this paper.

5. 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. E. Catmull: *A Hidden-Surface Algorithm with Anti-Aliasing*, ACM Computer Graphics, **12**-3 (proc. of SIGGRAPH'78), 1978, pp. 6-11.
3. F. Crow: *The Aliasing Problem in Computer Generated Shaded Images*, Communications of the ACM, **20**-11, 1977, pp. 799-805.
4. F. Crow: *A Comparison of Antialiasing Techniques*, IEEE Computer Graphics and Applications, **1**-1, 1981, pp. 40-48.
5. J. Foley, *et al.*: *Computer Graphics, Principles and Practice* (2nd. ed.), Addison-Wesley, 1990, pp. 617-646, 693-695.
6. N. Max: *Antialiasing Scan-Line Data*, IEEE Computer Graphics and Applications, **10**-1, 1990, pp. 18-30.
7. A. Oppenheim, A. Willsky, Y. Young: *Signals and Systems*, Prentice Hall, 1983, pp. 527-531, 544-545.
8. M. Pitteway, D. Watkinson: *Bresenham's Algorithm with Grey Scale*, Communications of the ACM, **23**-11, 1980, pp. 625-626.
9. G. Wolberg: *Digital Image Warping*, IEEE Computer Society Press, 1990, pp. 8-9, 106-111, 163-186.

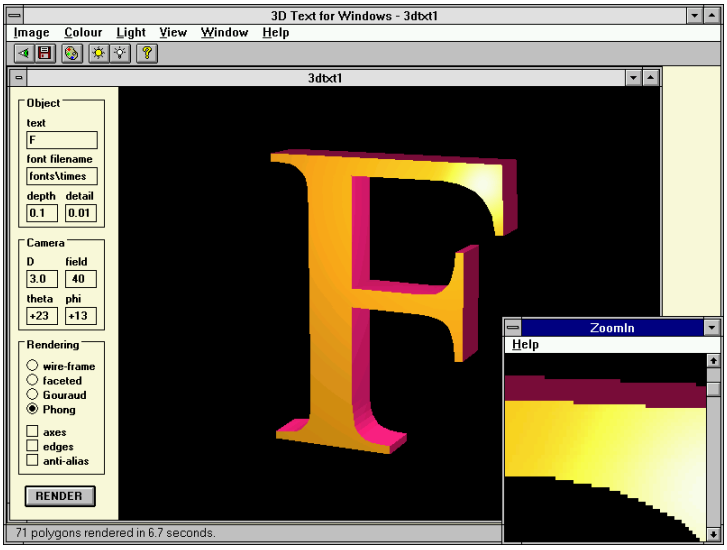

Figure 5: Aliased “F”.

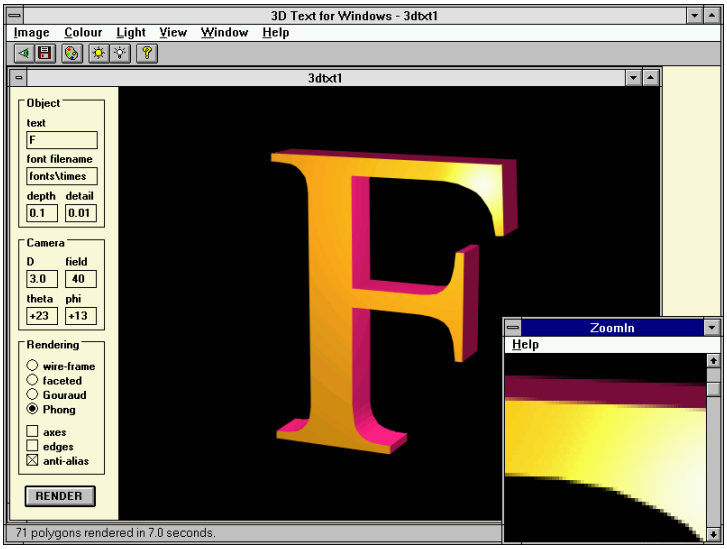

Figure 6: Anti-aliased “F”.