The A-buffer, an Antialiased Hidden Surface Method

Loren Carpenter

Computer Graphics Project
Computer Division
Lucasfilm Ltd

Abstract

The A-buffer (anti-aliased, area-averaged, accumulation buffer) is a general hidden surface mechanism suited to medium scale virtual memory computers. It resolves visibility among an arbitrary collection of opaque, transparent, and intersecting objects. Using an easy to compute Fourier window (box filter), it increases the effective image resolution many times over the Z-buffer, with a moderate increase in cost. The A-buffer is incorporated into the REYES 3-D rendering system at Lucasfilm and was used successfully in the "Genesis Demo" sequence in Star Trek II.

CR CATEGORIES AND SUBJECT DESCRIPTORS: I.3.3 [**Computer Graphics**]: Picture/Image Generation - Display algorithms; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - Visible line/surface elimination.

GENERAL TERMS: Algorithms, Experimentation.

ADDITIONAL KEY WORDS AND PHRASES: hidden surface, image synthesis, z-buffer, a-buffer, antialiasing, transparency, supersampling, computer imagery.

1. Introduction

There are many hidden surface techniques known to computer graphics. A designer of a 3-D image synthesis system must balance the desire for quality with the cost of computation. The A-buffer method, a descendant of the well-known Z-buffer, has proven to deliver moderate to good quality images at moderate cost. At each pixel, sufficient information is available to increase the effective resolution of the image several times over that of a simple Z-buffer.

2. Historical Perspective

The A-buffer belongs to the class of hidden surface algorithms called "scanline". The REYES (Renders Everything You Ever Saw) system, of which the A-buffer is a part, is a scanline renderer, but scanline order is not required by the A-buffer.

The first scanline algorithms[7] did perspective, clipping, sorting, visibility determination, and "filtering" all at the same time. They resolved visibility at one point per pixel, and aliased terribly, although our standards were different then. In 1974, E. Catmull described the Z-buffer method[2]. A Z-buffer is a screen-sized array of pixels and Z's. Objects, in no particular order, are examined to determine which pixels they cover. At each covered pixel, the perspective Z depth of the object is determined and compared with the Z in the array. If the new Z is closer, then the new Z, and the object's shade at this point, replaces the array's Z and pixel. This development started the trend toward modularizing the rendering process, as a Z-buffer could comprise the visibility section of almost any kind of renderer. Although extremely fast and simple, the Z-buffer aliases too much and cannot render transparent objects correctly.

The aliasing problems of the Z-buffer can be softened somewhat by modifying it from a point sampler to a line sampler so that visibility is determined over horizontal segments of scanlines[1]. In this way the line Z-buffer is very similar to the classical polygon algorithms of Watkins and others[7]. Polygons are sliced horizontally as in Watkins, but no X sorting is done. Instead, polygon segments conditionally overwrite others based on Z depth. The segment boundaries do not have to be coincident with pixel boundaries. This added information clears up aliasing of nearly vertical edges. However, nearly horizontal edges still alias and dropouts of small objects still occur.

In 1978, E. Catmull introduced the "ultimate" visibility method[3], a full polygon hidden surface process, based on Weiler-Atherton[8], at each pixel. Dropouts are precluded, as every sliver is accounted for. The color of the resulting pixel is simply the weighted average of all the visible polygon fragments. This can be extremely expensive. It is so expensive that it's primary use is in 2-D animation of a few fairly large polygons. In that application, most pixels are completely covered by some polygon, where the hidden surface process has a trivial solution. Pixels needing the full power of the visibility resolver are rare, and so the total cost per frame is acceptable.

3. Goals and Constraints

The visibility techniques described above span a wide range of computational expense and image quality. What is needed is a method that combines the simplicity and speed of the Z-buffer with the two dimensional anti-aliasing benefits of Catmull's full polygon process at each pixel.

The method must support all conceivable geometric modeling primitives: polygons, patches, quadrics, fractals, and so forth. It must handle transparency and intersecting surfaces (and transparent intersecting surfaces). It must do all this while being fast enough for limited production using a DEC VAX 11/780.

4. Strategy

The rendering system (REYES) in which the visibility processor was to reside began to take shape in mid 1981. Adaptive subdivision[5] (splitting geometric primitives until "flat" *on the screen*) would produce a common intermediate form: polygons. Everything would be converted to polygons in approximately scanline order, as the picture developed. The polygons would be thrown away after the visibility resolver had finished with them and their memory space would be used for polygons to be created later. To reduce the scope and complexity of the visibility resolver, polygons would be clipped to pixel boundaries. The visibility resolver would only have to deal with one pixel at a time.

In a virtual memory computer, like the VAX, code space is not a serious limitation, so it was decided to optimize the algorithm for the common cases and write potentially voluminous code for the unusual situations.

5. Geometry inside the pixel

The geometric information inside a complex pixel is vital to the correct display of the pixel. Pictures produced by REYES had to be free of aliasing artifacts. The aliasing deficiencies of the simple Z-buffer precluded its use. More resolution inside the pixel was called for, but a full polygon intersector/clipper was too expensive. After some experimentation, a 4x8 bit mask (figure 1) was selected to represent the subpixel polygons. Clipping one polygon against another becomes a simple boolean operation. The mask is similar in several ways to the mask of Fiume, Fournier and Rudolph[4], although both were developed independently.

Silhouettes of objects still exhibited coarse intensity quantization effects, so the actual screen area of subpixel-sized polygons was kept with the mask. Whenever possible, the actual area is used instead of the bit count in the mask.

6. The A-buffer Algorithm

The A-buffer works with two different data types: "pixelstructs" (distinct from pixels) and "fragments". A pixelstruct is two 32-bit words (figure 2), one containing a Z depth and the other either a color or a pointer. A fragment (figure 3) is for the most part a polygon clipped to a pixel boundary. Pixelstructs occur in an array the size and shape of the final image (like the Z-buffer). In REYES, the array is paged in software to save virtual memory space. If a pixel is simple, i.e. completely covered, the Z value is positive and the pixelstruct contains a color. Otherwise, the Z value is negative and the pointer points to a list of fragments sorted front-to-back by frontmost Z.

```
o o o o o o o o
o o o o o o o o
o o o o o o o o
o o o o o o o o
```

Figure 1. Pixel bit mask.

```
float          z;          /* negative Z */
fragment_ptr   flist;      /* never null */

               (or)

float          z;          /* positive Z */
byte           r, g, b;    /* color */
byte           a;          /* coverage */
```

Figure 2. Pixelstruct definition.

```
fragment_ptr   next;
short_int      r, g, b;        /* color, 12 bit */
short_int      opacity;        /* 1 - transparency */
short_int      area;           /* 12 bit precision */
short_int      object_tag;     /* from parent surface */
pixelmask      m;              /* 4x8 bits */
float          zmax, zmin;     /* positive */
```

Figure 3. Fragment definition.

The following discussion contains several symbols which we define here:

M	4x8 bit mask
A	area (0..1)
C	color (r, g, b)
Opacity	1 - transmission fraction
α	coverage, usually area times opacity[6]

Sorting in Z is necessary for two reasons. Proper calculation of transparency requires all visible transparent surfaces to be sorted in Z. The other benefit of a Z-sort is that fragments from the same geometric primitive tend to cluster together in the list and so can be merged. For example, a bicubic patch may be turned into several polygons. These polygons are all from the same continuous parent surface, but they may be chopped into fragments in an unpredictable order (depending on screen orientation, etc.) (figure 4). Merging two or more fragments simplifies the data structure and reclaims the space used by the merged-in fragments. If the result is opaque and completely covers the pixel we cannot with certainty reclaim hidden fragments, as they may be part of an incomplete intersecting surface.

The process of merging fragments is fairly straightforward. Fragments are merged if and only if they have the same object tag and they overlap in Z. This test is performed whenever a new fragment is added to a pixelstruct list. Object tags are integers assigned to continuous non-self-intersecting geometric primitive objects, like spheres and patches. The tag is augmented by a bit indicating whether the surface faces forward or backward, so as to prevent improper merging on silhouettes. If the fragments do not overlap on the screen ($\mathbf{M}_1 \cap \mathbf{M}_2 = \emptyset$) then the bitmasks are or'ed, the colors blended

$$\mathbf{C} = \mathbf{C}_1 \times \mathbf{A}_1 + \mathbf{C}_2 \times \mathbf{A}_2$$

and the areas added. If they overlap (which is highly abnormal), they are split into three parts.

$$\mathbf{M}_{\text{front-only}} = \mathbf{M}_{\text{front}} \cap \sim\mathbf{M}_{\text{back}}$$

$$\mathbf{M}_{\text{back-only}} = \mathbf{M}_{\text{back}} \cap \sim\mathbf{M}_{\text{front}}$$

$$\mathbf{M}_{\text{overlap}} = \mathbf{M}_{\text{front}} \cap \mathbf{M}_{\text{back}}$$

The contribution of the front fragment is computed,

$$\alpha_{\text{front}} = \mathbf{A}_{\text{front-only}} + \mathbf{Opacity}_{\text{front}} \times \mathbf{A}_{\text{overlap}}$$

the colors blended,

$$\mathbf{C} = \alpha_{\text{front}} \times \mathbf{C}_{\text{front}} + (1 - \alpha_{\text{front}}) \times \mathbf{C}_{\text{back}}$$

and the area computed.

$$\mathbf{A} = \mathbf{A}_{\text{front}} + \mathbf{A}_{\text{back}} \times \frac{\mathbf{A}_{\text{back-only}}}{\mathbf{A}_{\text{back-only}} + \mathbf{A}_{\text{overlap}}}$$

When no more fragments are to be sent to a pixelstruct, the pixelstruct's color is determined and written into the picture. Generally, the pixel will be fully covered by some object and a few pixel-sized fragments will remain. If any fragments are present, a recursive packing process is invoked.

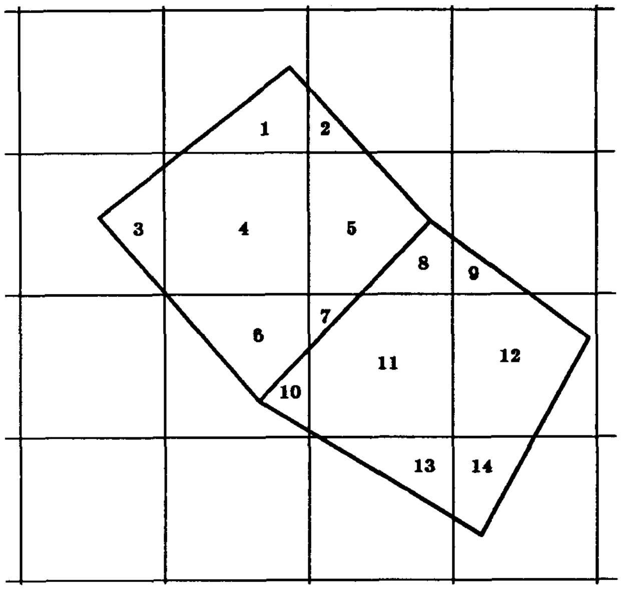

Figure 4. Typical fragment arrival order.

7. Packing fragments

Area-averaging means the color of a pixel is computed by the area-weighted average of the colors of the visible surfaces contained in the pixel. The problem is, then, how to determine the visible fragments and visible parts of fragments.

To understand the method used in the A-buffer, consider the following simplified example. Assume, for the moment, no transparency and no intersecting surfaces. If the fragment at the front of the list covers the pixel, we are done; otherwise, it covers part of the pixel. We divide the pixel into two parts, inside and outside, using the fragment's mask. The contribution of the inside part is the color of the fragment weighted by its area. The contribution of the outside part is some yet to be discovered color weighted by the complement of the fragment's area.

$$\mathbf{C} = \mathbf{C}_{\text{in}} \times \mathbf{A}_{\text{in}} + \mathbf{C}_{\text{out}} \times (1 - \mathbf{A}_{\text{in}})$$

The yet to be discovered color is found by recursively calling the packing routine with the outside mask to represent the rest of the pixel and a pointer to the next item in the fragment list.

We can now describe the method in more detail. We start the packing process with a full 32 bit search mask to represent the entire pixel. Fragments are considered only if they overlap the search mask. When all or part of a fragment is found within the search mask, the search mask part of the pixel is partitioned using the fragment mask.

$$\mathbf{M}_{\text{in}} = \mathbf{M}_{\text{search}} \cap \mathbf{M}_{\text{f}}$$

$$\mathbf{M}_{\text{out}} = \mathbf{M}_{\text{search}} \cap \sim\mathbf{M}_{\text{f}}$$

If $\mathbf{M}_{out} \neq 0$ we use a recursive call with $\mathbf{M}_{out}$ as the search mask to find the color of the rest of the searched area. If the fragment is transparent, a recursive call using $\mathbf{M}_{in}$ as a search mask is used to find the color of the surfaces behind the fragment to be filtered by the color of the fragment.

$$\mathbf{C}_{in} = \mathbf{Opacity}_f \times \mathbf{C}_f + (1 - \mathbf{Opacity}_f) \times \mathbf{C}_{behind}$$

The composite coverage is computed similarly.

$$\alpha_{in} = \mathbf{Opacity}_f \times \alpha_f + (1 - \mathbf{Opacity}_f) \times \alpha_{behind}$$

Otherwise, the color of the fragment suffices for $\mathbf{C}_{in}$. When we have the colors of the inside and outside regions we blend them weighted by their coverage.

$$\mathbf{C}_{returned} = \frac{\alpha_{in} \times \mathbf{C}_{in} + \alpha_{out} \times \mathbf{C}_{out}}{\alpha_{in} + \alpha_{out}}$$

For all but the first fragment on the list, we use the number of one bits in a mask to estimate area.

Now for intersections.

Pixels where intersecting surfaces are visible usually number in the dozens or hundreds in a typical 512x512 resolution picture. Also, the antialiasing along the line of intersection is not quite as critical as that on a silhouette, for example, because the contrast is often lower. These observations suggest we can get by with simple approximations.

Since no orientation information (vertices or plane equations) is kept in a fragment, we define an intersection to occur when the object tags differ and the fragments overlap in Z. This works satisfactorily in all but a few cases. Since we don't know exactly how much of the frontmost fragment is visible, we estimate it from the minimum and maximum Z values (figure 5).

$$\mathbf{Vis}_{front} = \frac{\mathbf{Zmax}_{next} - \mathbf{Zmin}_{front}}{(\mathbf{Zmax} - \mathbf{Zmin})_{front} + (\mathbf{Zmax} - \mathbf{Zmin})_{next}}$$

Since part of the front fragment obscures the next fragment and vice versa, we need to estimate the weighting factor to be used to blend the two fragment's colors.

$$\alpha_{in} = \mathbf{Vis}_{front} \times \mathbf{Opacity}_{front} + (1 - \mathbf{Vis}_{front}) \times (1 - \alpha_{next})$$

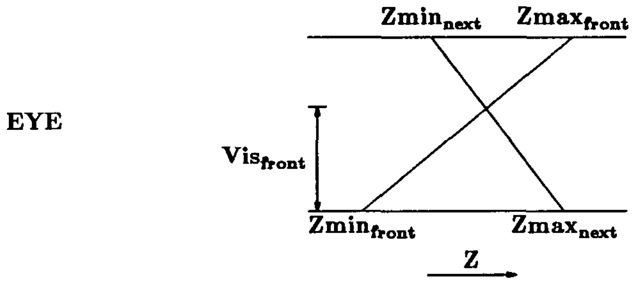

Figure 5. Visible fraction of front fragment.

```
Pack_under_mask (fragment_ptr, mask, r, g, b, a)

if this is the last fragment on the list
    return fragment's color and coverage
else
    find inside and outside masks
    if outside mask not empty
        find color and coverage of outside area
            (recursive call with outside mask)
    if fragment is transparent or overlaps in Z with next on list
        find color and coverage of what's behind
            (recursive call with inside mask)
    if nothing hidden behind the fragment affects its appearance
        return a blend of the fragment and the outside area
    else
        if Z's overlap with next fragment (maybe transparent)
            estimate visibility ratio
            estimate coverage of fragment
            blend fragment with what's behind it
            return blend of inside and outside
        else (just transparent)
            blend fragment with what's behind it
            return blend of inside and outside
end
```

Figure 6. Fragment packing procedure.

This is the sum of the unobscured part of the front fragment and the part of the front fragment filtered through the other fragment. Given these factors, we blend the front fragment with the other fragment within the inside mask.

$$C_{in} = \alpha_{in} \times C_{front} + (1 - \alpha_{in}) \times C_{next}$$

Then we blend the inside and outside part.

$$C_{returned} = \frac{\alpha_{in} \times C_{in} + \alpha_{out} \times C_{out}}{\alpha_{in} + \alpha_{out}}$$

A high level pseudocode description of the packer is given in figure 6.

8. Implementation details

The A-buffer is implemented in approximately 800 lines of C, including a substantial amount of debugging code. All arithmetic is done in fixed point (except for Z). There are two heavily used procedures inside the system that ought to be described in more detail.

The first is the bitmask constructor, which is designed to work correctly given arbitrary polygons. It begins with a polygon that has been clipped to a pixel boundary. The polygon bitmask is built up by exclusive or'ing together masks derived from the polygon's edges. Each polygon edge defines a trapezoid, bounded by the edge, the right side of the pixel, and the projection of the ends of the edge toward the right side of the pixel. (figure 7) The edge mask is constructed by or'ing together row masks taken from a table indexed by the quantized locations of the intercepts of the edge. The exclusive or of all these masks leaves one bits in the interior and zero bits elsewhere. All this sounds complicated, but it rarely involves more than eight boolean operations.

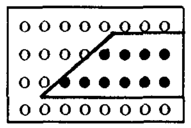

Figure 7. Polygon edge mask.

The other process computes the coverage ("area") of a polygon mask. Since the VAX has no bit counting instructions, the method is to strip off four bits at a time and look up the bit count in a table. The whole procedure can be put into a single C expression which generates efficient machine code.

9. Results

The REYES system, incorporating the A-buffer, has been used to make thousands of pictures. Figure 8 shows a magnified silhouette of the top of a teapot. Note the softness of the edge, even though the box filtering limits the edge intensity ramp to one pixel width. The Utah teapot, which appears in figures 8 and 9, is constructed so that its handle and spout penetrate its body. This is a common geometric modelling technique which avoids the explicit (and nearly intractable) calculation of the intersection curve. Figure 9 is a closeup of the upper part of the handle. The color of pixels through which the intersection curve passes is clearly a blend of the handle and body colors. Figure 10 is the "Genesis device". It is a collection of spheres, patches and polygons inside a partially transparent cylinder with quadrically modelled engines on the outside. Stars can be seen through the cylinder. All of figure 11, with the exception of the particle system grass plants, was rendered by REYES. The background of the picture was computed at 1024 lines and the foreground at 2048 lines resolution.

We have described a successful, relatively uncomplicated, anti-aliasing hidden surface mechanism. Like all visibility resolving methods, the A-buffer has its strengths, weaknesses, and limitations. It was designed to process the vast majority of pixels with minimum effort and maximum precision, spending compute time only on exceptional cases. On the other hand, the approximations used in the fragment intersection code can go astray if several surfaces intersect in the same pixel, and, of course, one cannot expect polygons smaller than the bitmask spacing to be sampled faithfully. Recognizing these limitations, we have found the A-buffer to be a practical, reliable means of producing synthetic images of high complexity.

References

1. CARPENTER, L., "A New Hidden Surface Algorithm," *Proceedings of NW76,* ACM, Seattle, WA, 1976.
2. CATMULL, E., *A Subdivision Algorithm for Computer Display of Curved Surfaces,* University of Utah, Salt Lake City, December 1974.
3. CATMULL, E., "A Hidden-Surface Algorithm with Anti-Aliasing," *Computer Graphics,* vol. 12, no. 3, pp. 6-11, ACM, 1978.
4. FIUME, E., A. FOURNIER, AND L. RUDOLPH, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," *Computer Graphics,* vol. 17, no. 3, pp. 141-150, ACM, July 1983.
5. LANE, J. M., L. C. CARPENTER, T. WHITTED, AND J. BLINN, "Scan-line methods for displaying parametrically defined surfaces," *Communications of the ACM,* vol. 25, no. 1, pp. 23-34, ACM, Jan. 1980.

6. PORTER, T. AND T. DUFF, "Compositing Digital Images," *Computer Graphics*, vol. 18, no. 3, ACM, 1984.

7. SUTHERLAND, I. E., R. F. SPROULL, AND R. A. SCHUMACKER, "A characterization of ten hidden-surface algorithms," *Computing Surveys*, vol. 6, no. 1, pp. 1-55, ACM, March 1974.

8. WEILER, K. AND P. ATHERTON, "Hidden Surface Removal Using Polygon Area Sorting," *Computer Graphics*, vol. 11, no. 3, pp. 214-222, ACM, 1977.

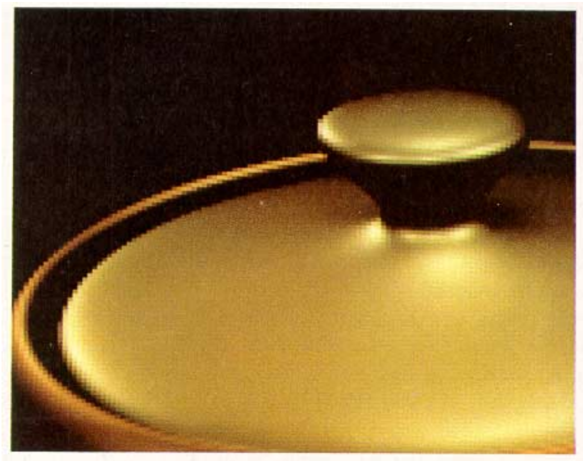
Figure 8. Detail of teapot silhouette. (4×)

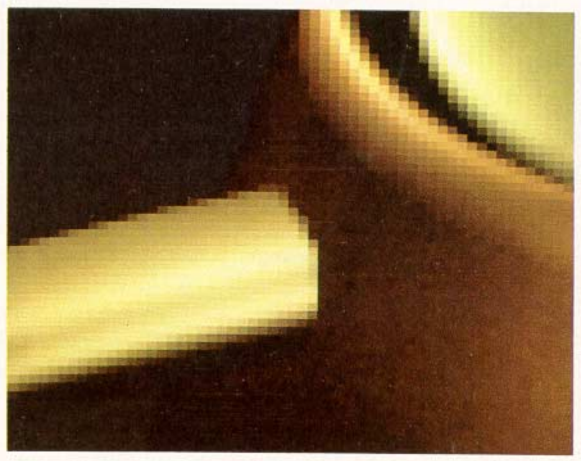
Figure 9. Detail of teapot handle intersection. (8×)

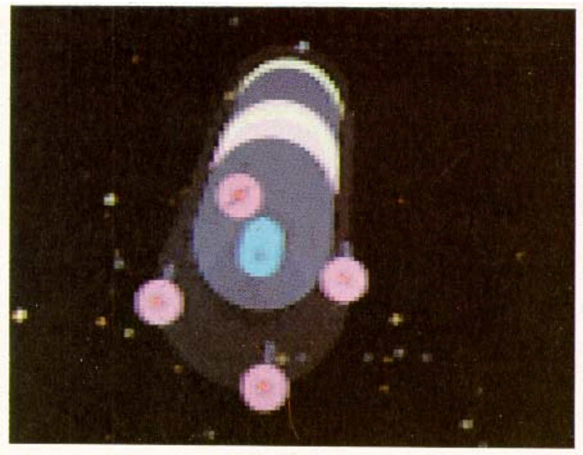
Figure 10. Genesis device. (4×)

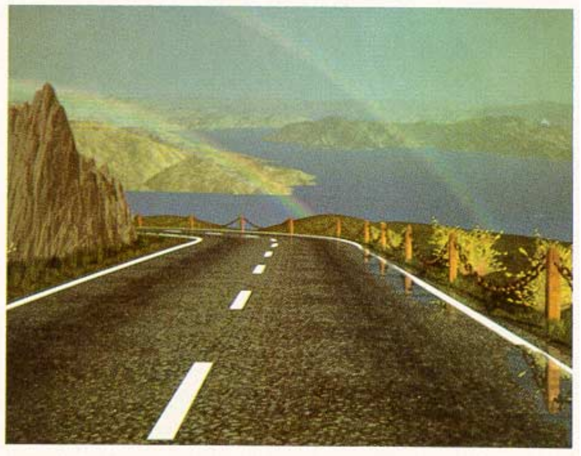
Figure 11. Road to Point Reyes.