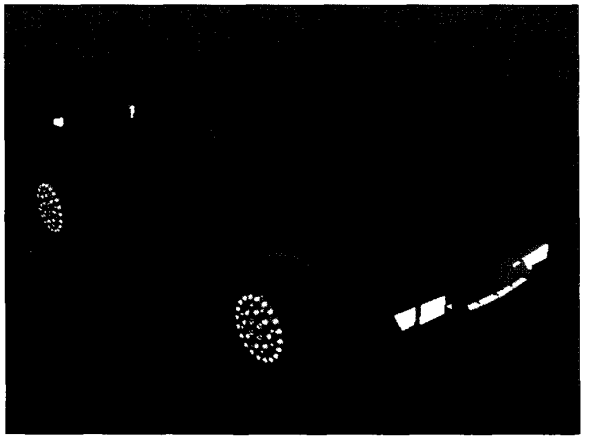
Rendering

Accurate Rendering by Subpixel Addressing

Olin Lathrop
Cognivision
David Kirk and Doug Voorhies
Hewlett-Packard



We present a simple method for eliminating most pixel-positioning errors when rendering lines and polygons with Bresenham's algorithm. Our method affects only the calculation of the initial values for Bresenham's vector-generating algorithm. It doesn't alter the actual vector-generating algorithm, requiring only integer arithmetic to find the next pixel in a vector.

Our method eliminates all dropouts and virtually all overlaps between adjacent polygons whose edges lie on the same line. This eliminates the need to "grow" polygons to avoid dropouts and opens the possibility of drawing surfaces composed of adjacent polygons with read/modify/write pixel operations, such as Add or alpha buffering. We also show that most rendering artifacts of today's display controllers ultimately result from pixel-positioning errors, not insufficient z-buffer resolution.

Most current commercial display controllers contain hardware that implements Bresenham's vector-generating algorithm.[1] This is an attractive algorithm for hardware implementation because it only requires simple integer arithmetic to create successive coordinates along a vector. When we express the vector endpoint coordinates in the integer pixel space, then the setup calculations also require only simple integer arithmetic. Thus, modest hardware can start with integer endpoint coordinates of a vector and address pixels without further intervention. The advantages of a total hardware implementation are clear, but the disadvantages are more subtle. Expressing vector endpoints in the integer-pixel coordinate space amounts to always drawing vectors that start and end on pixel centers. This produces pixel addressing errors of up to one half of a pixel, assuming the vector was originally defined in a continuous space (32-bit floating point is effectively continuous in this context).

We often render polygons in hardware by employ-

0272-17-16/90/0900-0045$01.00

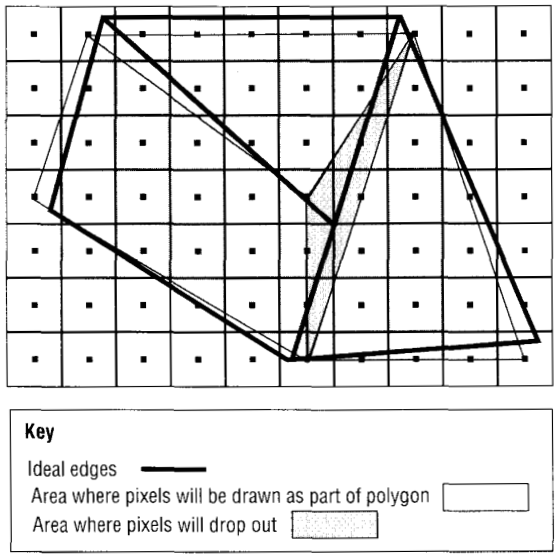


Figure 1. Dropouts or overlaps can easily occur when abutting polygons do not share vertices, and vertices are rounded to pixel centers.

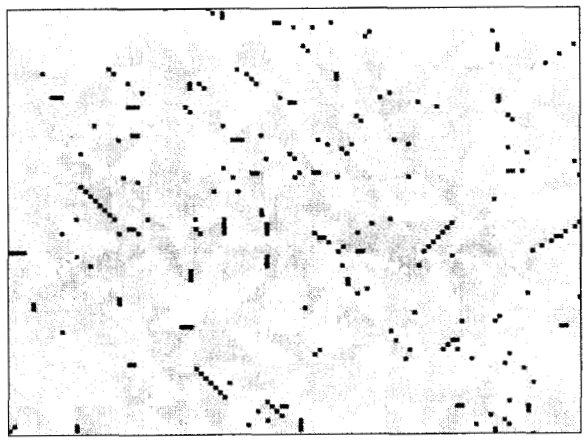


Figure 2. Many triangles abutting as shown in Figure 1. The dark blue pixels are dropouts, and the yellow pixels are overlaps.

ing Bresenham vector generators to walk the edges. The position of such polygon edges, therefore, can be in error up to one half of a pixel. Although generally not noticeable as an absolute positioning error, this can cause visible dropouts or overlaps between polygons that abut and do not share the same vertices. A common solution has been to grow polygons by at least one half of a pixel before rounding the vertices to the nearest pixel centers. This guarantees that we won't miss any pixels at the boundary between two adjacent polygons. It also implies, however, that many of the pixels near polygon boundaries are written more than once. This generally precludes drawing polygonal meshes with read/modify/write pixel operations, which are useful for blending effects such as transparency.

As we show later, pixel position errors create further errors in the colors and z. These errors result in ragged color contours, ragged intersection edges, and z-buffer "poke-throughs."

In this article, we describe a method that solves all these problems by eliminating the root cause: errors in choosing which pixels should be written when rendering a polygon.

Dropouts and overlaps

As discussed above, as a result of positioning errors, some pixels are not written when they should be (dropouts), and some pixels are written more than once, even though they were inside only one polygon (overlaps). These artifacts are not apparent when we render adjacent polygons that have exactly the same vertices on adjacent edges. As we draw each polygon, the shared-edge endpoints will be rounded to pixel centers. Although we might draw the edge with some error, no dropouts or overlaps will occur because the identical edge was chosen for each polygon. This is not the case, however, when one polygon is adjacent to more than one other polygon along the same edge, as illustrated by Figure 1.

Figure 1 shows a grid of pixels, with the center of each pixel marked as a black square. The bold lines show the edges of three polygons as defined in continuous space. The lightly shaded regions show the polygons after we snapped their vertices to pixel centers. The dark triangle represents a region where dropouts will occur if any pixel centers fall there. This type of polygon abutting can occur quite frequently, especially when the polygons result from tesselating an object with adaptive subdivision or when separate meshes meet.

Figure 2 is an example of many polygons abutting as shown in Figure 1, rendered without the subpixel algorithm. We made it by first constructing a rectangular grid of points, with small random offsets applied to interior grid points. We broke the quadrilaterals

between the grid points into two triangles. Every triangle then had a 50 percent probability of being subdivided into four smaller triangles. We originally cleared the image to a blue background color. Then we arithmetically added all the triangles into the pixel grid with a color chosen to result in a randomly chosen shade of gray (after the addition). As diagrammed in Figure 1, this procedure created many examples of dropouts and overlaps. Dropouts show the dark blue background color, and overlaps show up as yellow.

Shading and z inaccuracies

Most of today's display controllers that can render polygons in hardware also interpolate color and z values across the polygon. Problems arise in extrapolating a color value for pixels that lie outside the polygon. Growing polygons to eliminate dropouts in favor of overlaps exacerbates these problems.

If we base the interpolation on the original polygon definition, then the interior pixels will be set to the correct values. However, even with linear interpolation, the exterior pixel values can exceed the range of all the vertex values. The pixel values may exceed the maximum or minimum representable values. For 8-bit pixels, clamping to 0 and 255 can help somewhat, but this has two problems. First, the slopes of interpolated values can be very large, and an external pixel value may exceed the hardware's ability to detect an out-of-range value. Second, we may have set vertex values deliberately to use only a portion of the color range. This is common in pseudocolor applications or when sharing a lookup table between multiple windows. The hardware rarely has knowledge of what we consider the "limits" of the color space.

To avoid these problems, the interpolation is usually adjusted so that the original vertex values appear at the vertices of the polygon actually drawn. This guarantees that the interpolated values will remain bounded (assuming linear interpolation), at the cost of adding a potential error to all the pixels. If we enlarge the polygon by one half of a pixel as discussed earlier (allowing edges to be up to one pixel from the correct position), then we can think of the pixel value as the result of point sampling the polygon with up to one pixel of displacement noise added to the sample-point coordinate. Therefore, the overall effect on the pixel value can be quite large, depending on the slope of the interpolated value with respect to x and y. Despite this error, we usually consider this the least objectionable solution, and most commercial systems employ it.

This error on the color values can cause problems when we use a restricted range of pseudocolors or when we deliberately create contours. For full color images, it is generally unnoticeable. However, the effects of this error on the z values can result in more severe artifacts. Since any one z value can represent a depth from up to one pixel away, the intersection of any two surfaces can have an error of at least one pixel in either direction from the ideal intersection curve. The potential error increases as the two surfaces intersect at more shallow angles. The resulting artifacts usually appear as "ragged" intersections or poke-through.

Evidence

Figure 3 shows a 512×410 image made with software using a 16-bit z-buffer. We used floating-point calculations to determine which pixels to draw for each polygon. This is the control picture and is con-

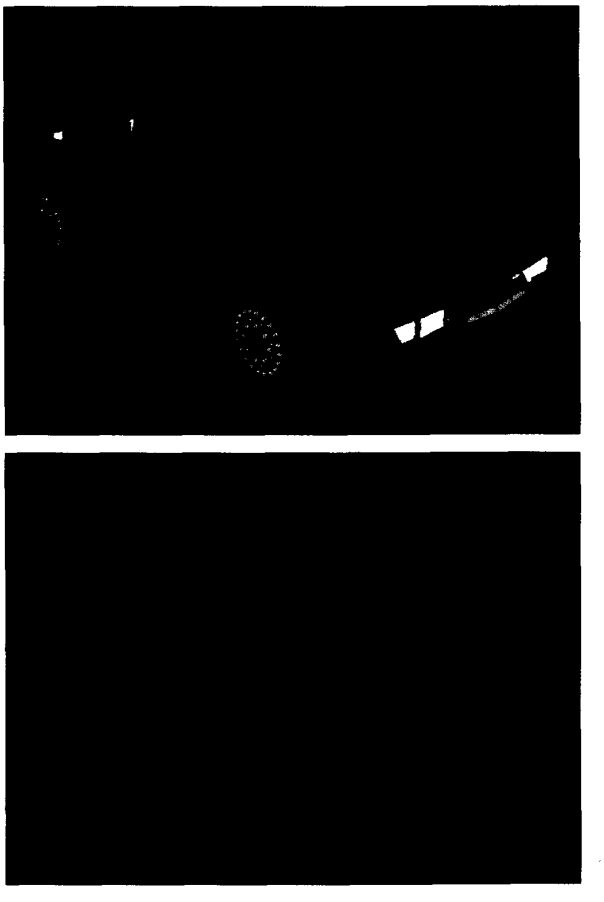


Figure 3. "Correct" reference image. We made floating-point calculations to find pixel positions. The z-buffer width is 16 bits.

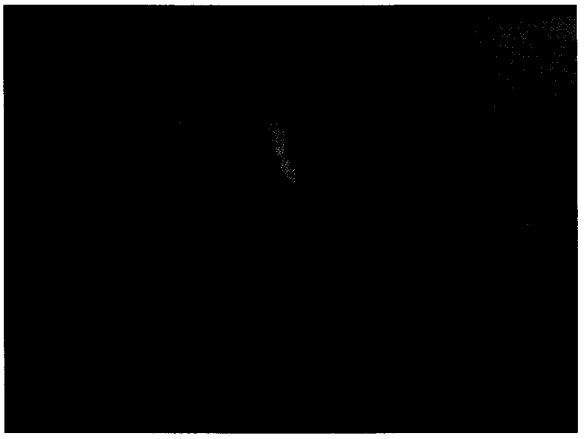


Figure 4. Rendered on an Apollo DN590-T using Apollo's 3D GMR graphics library. (Newer Apollo systems employ subpixel addressing and do not exhibit these artifacts.)

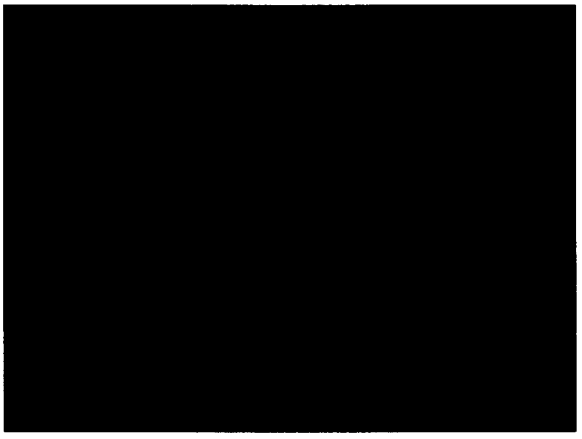


Figure 5. This was originally rendered on Raster Technologies Model One/380. This hardware was no longer available to us, so we could not rerender the image with exactly the same viewing parameters and background color. The checkerboard pattern results from the "screen door" technique of rendering the windshield semi-transparent.

sidered to be "correct." The data consists of about 67,000 triangles.

We created Figures 4 and 5 on commercial display controllers that use traditional integer set-up for Bresenham's algorithm. We have verified this problem on current workstations. Note the poke-through, particularly visible at the back of the driver's seat, and the ragged edges around the windshield.

Figure 6 is a software simulation of this problem. We grew the vertex for each triangle outward from the triangle center by one fourth of a pixel, then rounded the vertex to the nearest pixel center. Note the similarity of the artifacts with those in Figures 4 and 5.

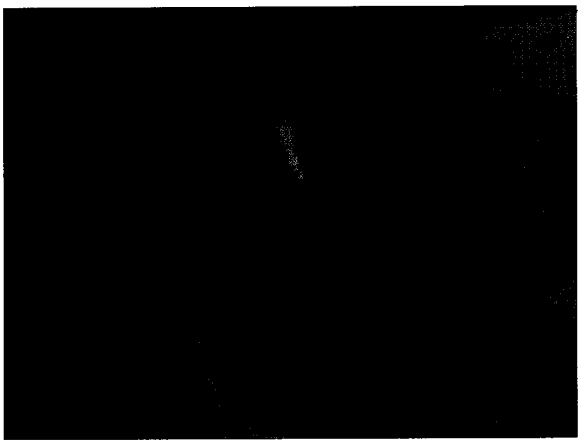


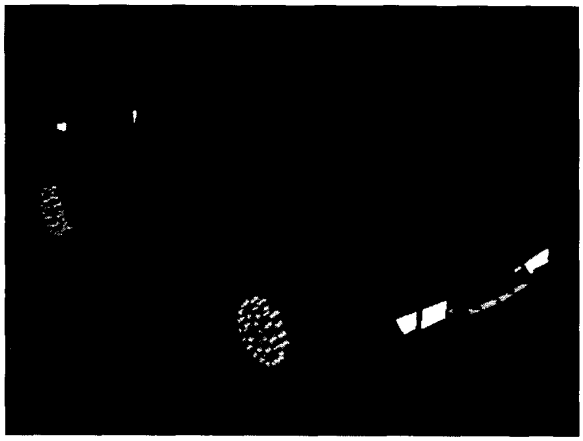
Figure 6. Software simulation of artifacts observed in Figures 4 and 5. Each triangle vertex was grown outward by one fourth of a pixel, then rounded to the

A common misconception is that ragged edges are caused by insufficient z resolution. This is usually not the case. A 16-bit z-buffer covering a 16-foot range provides about 75 micrometers of resolution. (This is about half the thickness of the average bumper sticker.) We rendered Figure 7 in the same way as Figure 3, except that we limited the z-buffer to only 6 bits. While many large and obvious artifacts result from the limited z resolution, the figure has no ragged edges of the type found in Figures 4, 5, and 6.

The subpixel addressing algorithm

We will now derive the algorithm for rasterizing a vector in the first quadrant. Allowing the vector to be in the first quadrant instead of the first octant permits the minor axis to take a large step while the major axis advances exactly one pixel. For simplicity, we won't show the terminating conditions. The algorithm below will walk a ray with a given starting point and slope.

In our implementation, the software breaks polygons into trapezoids with horizontal tops and bottoms, and the hardware uses the subpixel algorithm to walk along the sides in positive y. The software draws pixels along each scan line from one side to the other.



nearest pixel center. The artifacts got progressively worse as the growth amount increased in the range from zero to one half of a pixel.

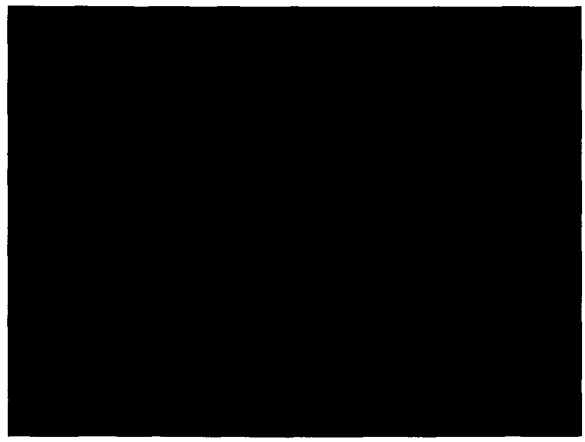


Figure 7. Rendered using subpixel addressing, but with only a 6-bit z-buffer. Note the lack of ragged edge artifacts found in figures 4, 5, and 6.

This increases efficiency because the image memory implementation is optimized for writing horizontally. Therefore, in our implementation for polygons, *y* is always the major axis, and it is always walked in the positive direction.

Figure 8 illustrates the geometry involved. We chose the coordinate system so that pixel centers are at fractional coordinates of one half. Therefore, only a Trunc operation is required to find the integer coordinates of a pixel given any point within it.

Maj_s, Min_s is the starting coordinate of the vector. Dmaj and Dmin are the major and minor axis lengths of the vector. Maj_0, Min_0 is the vector coordinate where it intersects the first major axis pixel center. It is useful to find it first, since this is essentially the "starting coordinate" for the rest of the algorithm:

$$Maj_0 = \text{round}\,(Maj_s) + 1/2$$
$$Min_0 = (Dmin/Dmaj)(Maj_0 - Maj_s) + Min_s$$

We will start with a slope fraction algorithm, then evolve the inner loop into Bresenham's algorithm. The slope fraction algorithm is

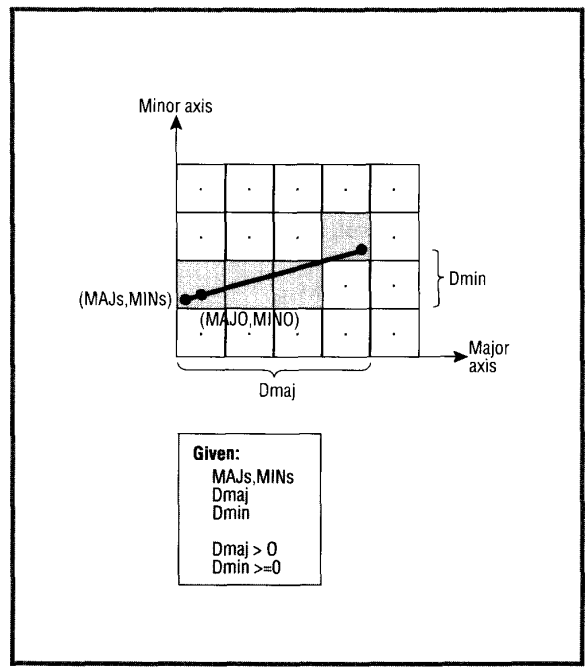


Figure 8. The original line starts at (Maj_s, Min_s). Our algorithm will start at the first major axis crossing, which is (Maj_0, Min_0).

Imaj <– – Trunc(Maj_0)	**(1a)**
Min <– – Min_0	**(2a)**
loop Imin <– – Trunc (Min)	**(3a)**
draw (Imaj, Imin)	**(4a)**
Imaj <– – Imaj + 1	**(5a)**
Min <– – Min + Dmin/Dmaj	**(6a)**
Repeat from #3a	**(7a)**

Imaj and Imin are the integer coordinates of the pixel that will be drawn each iteration. Min is the real number value of the minor axis where the cur-

rent major axis pixel center intersects the line. This algorithm will be altered by eliminating Min. A new value E will now track the displacement of Min inside the current pixel:

```
Imaj <-- Trunc (Maj0)                          (1b)
  Imin <-- Trunc (Min0)                        (2b)
  E <-- frac (Min0)                            (3b)
  loop  draw (Imaj, Imin)                      (4b)
        Imaj <-- Imaj + 1                      (5b)
        Imin <-- Imin + Trunc (Dmin/Dmaj)      (6b)
        E <-- E + frac(Dmin/Dmaj)              (7b)
  if E >= 1 Then                               (8b)
    Imin <-- Imin + 1                          (9b)
    E <-- E - 1                                (10b)
  Repeat from #4b                              (11b)
```

In step 6b, Imin is only incremented by the minimum number of whole pixel steps it must take each cycle. Any additional fractional displacement inside the pixel accumulates in E in step 7b. If the total accumulated fraction amounts to another whole pixel, then this quantity is "transferred " into Imin in steps 9b and 10b. Now eliminate the potential for two successive adds into E in the same cycle (steps 7b, and 10b) by adding one of two possible values into E, depending on the result of the decision.

```
      Imaj <-- Trunc (Maj0)                    (1c)
      Imin <-- Trunc (Min0)                    (2c)
E <-- frac (Min0) + frac (Dmin/Dmaj)           (3c)
loop  draw (Imaj, Imin)                        (4c)
      Imaj <-- Imaj + 1                        (5c)
      if E >= 1                                (6c)
        Then                                   (7c)
          Imin <-- Imin + Trunc (Dmin/Dmaj) + 1
                                               (8c)
          E <-- E + frac (Dmin/Dmaj) - 1       (9c)
        Else                                   (10c)
          Imin <-- Imin + Trunc (Dmin/Dmaj)    (11c)
          E <-- E + frac (Dmin/Dmaj)           (12c)
      Repeat from #4c                          (13c)
```

Now move the decision threshold for E down from one to zero by adjusting the initialization for E. Also note that the values added to Imin and E in the loop are constants. We'll give these expressions names defined before the loop. This will make it clear exactly how much computation is necessary inside the loop.

```
Imaj <-- Trunc (Maj0)                          (1d)
Imin <-- Trunc (Min0)                          (2d)
E <-- frac (Min0) + frac (Dmin/Dmaj) - 1       (3d)
DEA = frac (Dmin/Dmaj)                         (4d)
DEB = frac (Dmin/Dmaj) - 1                     (5d)
Astep = Trunc (Dmin/Dmaj)                      (6d)
Bstep = Astep + 1                              (7d)
loop  draw (Imaj, Imin)                        (8d)
Imaj <-- Imaj + 1                              (9d)
if E >= 0                                      (10d)
  Then                                         (11d)
    Imin <-- Imin + Bstep                      (12d)
    E <-- E + DEB                              (13d)
  Else                                         (14d)
    Imin <-- Imin + Astep                      (15d)
    E <-- E + DEA                              (16d)
Repeat from #8d                                (17d)
```

Note that the inner loop now has the same form as the standard Bresenham's algorithm, although E, *DEA*, and *DEB* are still real numbers. If the vector is restricted to the first octant (Dmin < Dmaj), and the starting coordinate is constrained to be on a pixel center Min0 = 1/2, then lines 3d-5d become

```
E <-- 1/2 + (Dmin/Dmaj) - 1                    (1e)
DEA = Dmin/Dmaj                                (2e)
DEB = (Dmin/Dmaj) - 1                          (3e)
```

Since the decision threshold for E is now at zero, E, *DEA*, and *DEB* can be multiplied by any positive scale factor without affecting the algorithm. If we apply a scale factor of 2 · Dmaj, then lines 1e, 2e, and 3e become

```
E <-- 2 · Dmin - Dmaj                          (1f)
DEA = 2 · Dmin                                 (2f)
DEB = 2 (Dmin - Dmaj)                          (3f)
```

This is the all-integer setup for the standard Bresenham's algorithm. It verifies that the algorithm in 1d-17d is truly a superset of Bresenham's algorithm.

We must not constrain the desired algorithm to start on pixel centers, but E, DEA, and DEB must still become integers to allow for easy hardware execution. We can do this by applying a scale factor to equations 3d-5d. We choose the scale factor to maximize the number of bits used in the integer registers in the hardware. For 16-bit registers, this means scaling the numbers so that the largest magnitude is always from 16,384 to 32,767. E is always below DEA and above DEB, DEA is always positive, and DEB is always negative. Furthermore, DEA is always exactly DEB + 1. Therefore (assuming 16-bit integer hardware), the

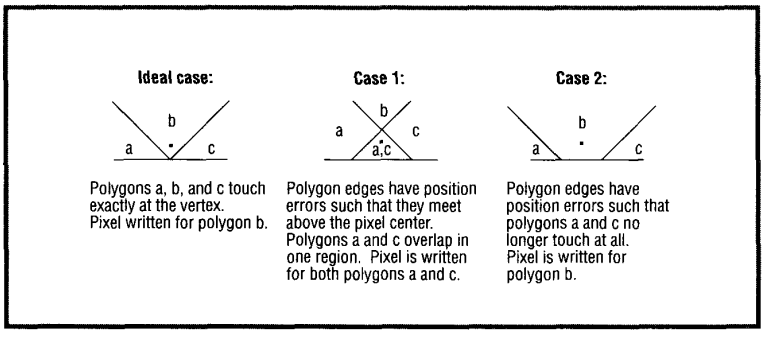


Figure 9. Each illustration shows three polygons meeting at a common vertex near the pixel center, shown as the dot. A potential overlap can occur when three polygons meet just below a pixel center. The two edges between adjacent polygons can have position errors as shown. Note that it is not possible to drop the pixel.

final algorithm with integer *E*, DEA, and DEB is 1d-17d, with lines 3d-5d altered to yield

	Imaj < – – Trunc (Maj_0)	**(1g)**
	Imin < – – Trunc (Min_0)	**(2g)**
	E < – – round ((frac (Min_0) + frac ($Dmin/Dmaj$) – 1) · 32767)	**(3g)**
	DEA = round (frac($Dmin/Dmaj$) · 32767)	**(4g)**
	DEB = DEA – 32767	**(5g)**
	Astep = Trunc ($Dmin/Dmaj$)	**(6g)**
	Bstep = Astep + 1	**(7g)**
loop	draw (Imaj, Imin)	**(8g)**
	Imaj < – – Imaj + 1	**(9g)**
	if E > = 0	**(10g)**
	Then	**(11g)**
	Imin < – – Imin + Bstep	**(12g)**
	E < – – E + DEB	**(13g)**
	Else	**(14g)**
	Imin < – – Imin + Astep	**(15g)**
	E < – – E + DEA	**(16g)**
	Repeat from #8g	**(17g)**

Remaining errors

Using round functions (equations 3g, 4g), we see that this is not an exact solution, although the position error is a small fraction of a pixel. However, these positioning errors do not cause the dropout problem illustrated in Figure 1, because *DEA* and *DEB* only depend on the slope of the ideal line. Since all colinear line segments have the same slope, they will produce the same integer values of *DEA* and *DEB*. This repeatability is far more important in reducing artifacts than absolute accuracy. Once we select *DEA* and *DEB*, at least half the full range of *E* will map to a one-pixel sideways displacement of the line. For 16-bit *E*, this implies a resolution of about 1/16,000 pixel.

Errors at pixel centers

Even though the positioning errors are very small, they can cause the wrong decisions when a line passes within the error tolerance of a decision point. Such decision points exist at every pixel center. As discussed above, an occasional error when an edge passes within about 1/16,000 of a pixel center is irrelevant because the same error repeats for the polygon on each side of the line. However, if a vertex of three or more polygons lies within an error tolerance of a pixel center, the algorithm may write the pixel more than once. Figure 9 illustrates this.

The ideal vertex must lie below the pixel center for this case to occur. In our implementation, the software decides how many scan lines are in a polygon and does not tell the hardware to draw the scan line if the ideal vertex lies above the pixel center. Because of this, the error is guaranteed to be an overlap instead of a dropout.

With randomly placed vertices, this error would occur so infrequently that we could totally ignore it. Unfortunately, distribution of vertices is not random,

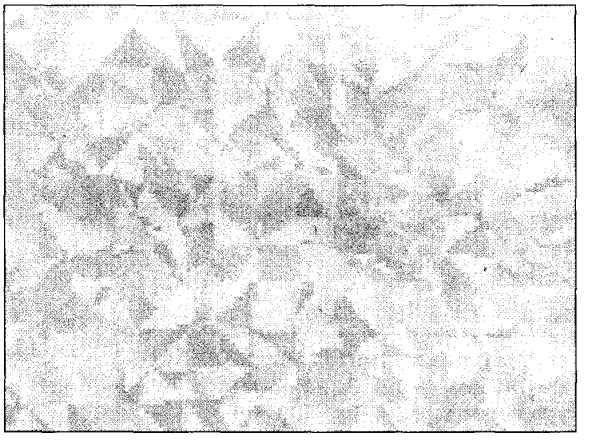


Figure 10. We used the same data as for Figure 2, but rendered with the subpixel addressing algorithm. Note the lack of blue dropouts or yellow overlaps. This indicates that every pixel of the tiled region was written exactly once, despite the occurrence of many cases as diagrammed in Figure 1.

and vertices can frequently occur on pixel centers. This is especially likely for 2D applications. Our solution is to add a small offset of about one 1/100 pixel to all vertices when they are transformed into pixel-grid space. The region in which vertices are unusually likely to occur is then moved far away from pixel centers.

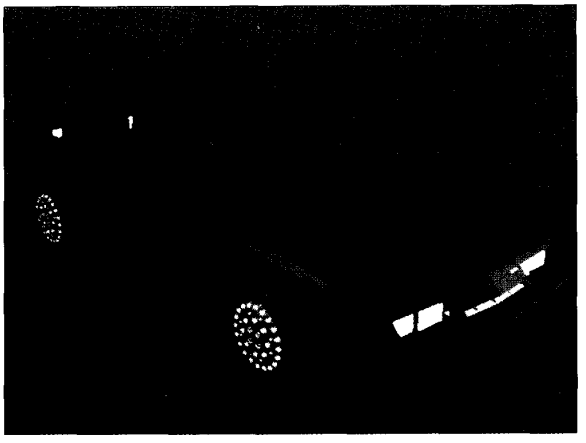
Results

Figure 10 shows the same data as Figure 2, except that we used the subpixel algorithm. This confirms that we have solved the dropout problem as diagrammed in Figure 1. Figure 11 shows subpixel positioning applied to the Chrysler Laser. These images appear identical to the controls in Figure 3.

We implemented our subpixel addressing algorithm on the Apollo DN10000VS. Its success provides further proof that the algorithm is both useful and practical. ■

Acknowledgment

We performed the work described here while employed by Apollo Computer.



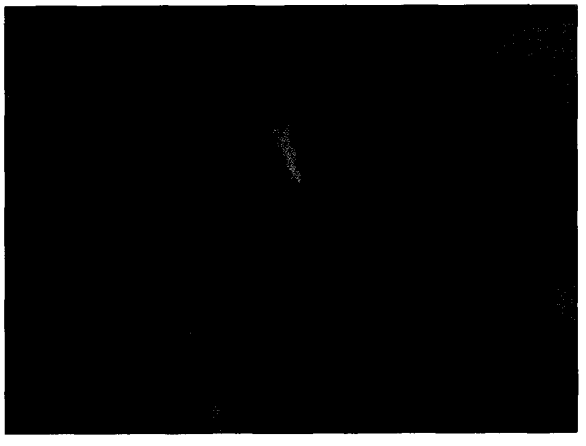
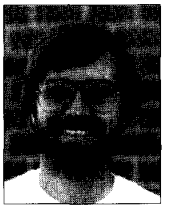


Figure 11. The subpixel algorithm applied to the test case data. Compare this to the control in Figure 3.

Reference

1. Bresenham, J.E., "Algorithm for Computer Control of a Digital Plotter," *IBM Systems J.*, Vol. 4, No. 1, Jan. 1965, pp. 25-30.



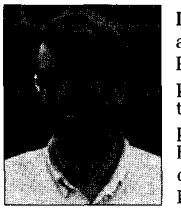
Olin Lathrop is one of the two founders of Cognivision, a company specializing in data visualization software and services. He is the vice president of research at Cognivision, and his current interests include visualization algorithms and techniques and how to best present information for human understanding. Lathrop is an occasional lecturer on computer graphics and data visualization at conferences and universities, but he also enjoys teaching the basics to newcomers in the field.

Lathrop previously worked at Apollo Computer on the graphics subsystem architecture of the DN10000VS workstation, and at Raster Technologies he designed 3D graphics products such as the model One/380 and the model One/25-S. He is a member of ACM.

Lathrop received his BS and his MEng in electrical engineering from Rensselaer Polytechnic Institute.

David Kirk is an AT&T fellow working toward a PhD in the California Institute of Technology Computer Graphics Group. He also works for Apollo Computer, the recently acquired Apollo Systems Division of Hewlett-Packard. He was one of the architects of the Apollo DN10000 graphics system and is currently part of the RISC Technology Group. His research interests include efficient ray tracing algorithms, realistic image synthesis, and parallel hardware and algorithms for computer graphics. Kirk has worked previously at the IBM Cambridge Scientific Center, at Computervision (as a CAD software developer), and at Raster Technologies, where he worked on the architecture of 3D graphics display controllers.

Kirk received his BS and MS in mechanical engineering from the Massachusetts Institute of Technology in 1982 and 1984, respectively. This year he received an MS in computer science from the California Institute of Technology.



Doug Voorhies is a senior consulting engineer at the Apollo Systems Division of Hewlett-Packard. His technical interests center on applying unusual hardware and software techniques to previously unsolved systems problems. At the Apollo Systems Division of Hewlett-Packard he led the architecture and design of the 3D graphics for Apollo's DN10000. Prior to that he designed a logic simulation engine at Apollo, designed CPUs for Prime Computer, and was a member of MITRE's technical staff.

Voorhies received a BS in electrical engineering from Yale University in 1969.

Lathrop can be contacted at Cognivision, 319 Littleton Road, Suite 100, Westford, MA 01886. Kirk and Voorhies are at Apollo Computer, 300 Apollo Way, Chelmsford, MA, 01824.