Behavior Research Methods, Instruments, & Computers
1989, 21 (1), 59-66

INSTRUMENTATION & TECHNIQUES

Antialiasing: A technique for smoothing jagged lines on a computer graphics image—an implementation on the Amiga

PETER P. TANNER, PIERRE JOLICOEUR, WILLIAM B. COWAN,
KELLOGG BOOTH, and FLYNN D. FISHMAN
University of Waterloo, Waterloo, Ontario, Canada

Images displayed on computer graphics displays often suffer from the presence of aliasing artifacts that give a jagged appearance to lines or polygon edges displayed on the screen. This paper details the problems associated with these artifacts and presents a method for drawing *antialiased* lines—ones in which the artifacts have been considerably reduced. The line-drawing routine is further developed to incorporate *gamma correction*, to take into account the nonlinear relationship between the intensity of the light emission from the phosphor of the monitor and the gray-scale values used to control the intensity on the screen.

Computer graphics displays are being used with increasing frequency to present images to subjects in psychological experiments. The advantages of such an approach over use of the more traditional slide-projector equipment are relatively apparent in terms of temporal control of the image, randomization of the image selection, and ease of setup. However, one disadvantage of computer-generated images, which can inappropriately bias experimental results, is less apparent: the presence of *aliasing artifacts* intrinsic to framebuffer computer graphics displays. These artifacts result in an image on the screen that differs from the image that the experimenter wishes to have displayed.

This paper presents a line-drawing technique, *antialiasing*, that reduces the aliasing problem (or *jaggies*). First, a simple line-drawing routine similar to those commonly available on personal computers is illustrated; the problems associated with it are also discussed. Second, a modified version that incorporates an antialiasing technique is presented. Finally, an inadequacy of this second routine is described—that is, its lack of *gamma correction*—and a third line-drawing routine is presented in order to rectify this inadequacy.

Aliasing

Raster display systems that use cathode-ray tube (CRT) displays, ink jet, electrostatic, or laser printers create images with arrangements of colored dots. The resulting images, therefore, are only approximate, apart from the rare case in which it is actually desired to create an image that really appears to be made up of dots. The quality of the approximation depends on the resolution of the screen, the resolution of the color information that can be stored for each dot, and the care taken to eliminate the effects caused by using this approximation of the desired image.

Figure 1 shows a collection of "straight lines" as drawn on the screen of a personal computer using Bresenham's algorithm, an algorithm typical of those used to draw lines

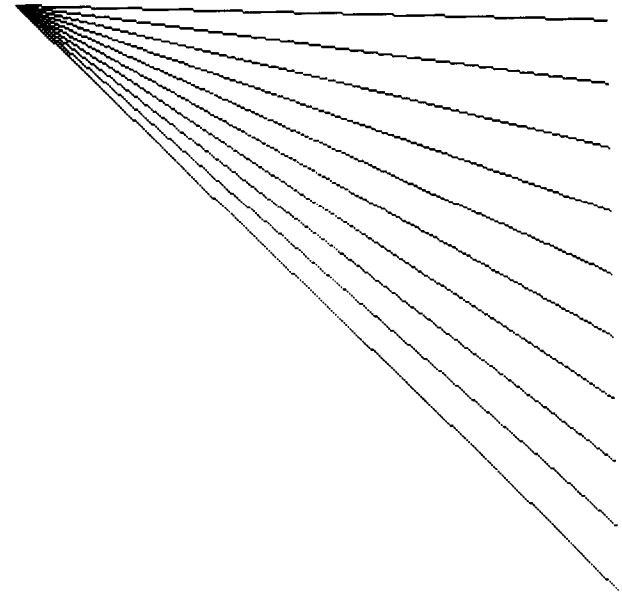

Figure 1. Lines drawn by an algorithm similar to those used in most displays. Note that all figures showing illustrations of lines have been produced with halftoning techniques rather than as photographs of the display, to avoid the loss of picture quality inherent in the printing process.

We thank Keith McGowan for his timely assistance. Correspondence may be addressed to Pierre Jolicoeur, Psychology Department, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

on computer displays (Bresenham, 1965). The jaggies (i.e., the staircase effect) visible on these lines are the result of drawing the line into a matrix of pixels with no attention given to line smoothing or antialiasing. The presence of jaggies has a number of consequences. First, it may give a viewer strong cues as to whether a line is perfectly horizontal or vertical or not. While this cue may be helpful in certain computer graphics applications, it is an artificial information aid whose presence is normally not intended by the experimenter. Second, it may reduce the subject's capacity to understand the image on the display. For example, Booth, Bryden, Cowan, Morgan, and Plante (1987) showed that, under certain conditions, there was a significant improvement in a subject's ability to count the number of local elements in an image when that image was processed to reduce the aliasing artifacts. Third, the location and number of jaggies change substantially when a line moves or changes orientation. A subject who views a rotating line receives significant cues that indicate the onset and speed of the rotation from the movement of the jaggies along the line. A subject comparing images A and B, where B is a rotated version of A, may well conclude that they are different images, because of the different positions of the steps on the lines within the image. Fourth, for objects that are small in relation to the pixel grid, the staircase artifacts can cause different parts of an image to change their spatial relationships. An example of this occurred in an air traffic simulation system in which distant airplanes, covering only a few pixels, would change shape as they moved. Their wings would drop from one row of pixels to the next at a different time than would the rest of the plane, leading to confusion as to whether it was a midwing plane or an upperwing plane (for another example, see Lindholm, 1988). Finally (although this list is not exhaustive), jaggies result in Moiré patterns (Crow, 1977) if there are several near-parallel lines drawn close together.

Antialiasing techniques make the aliasing artifacts in rendered images less noticeable. One such technique is to defocus the picture tube on which the image is displayed. Although this approach does work to some extent, it does so at the cost of a considerable reduction in the viewable resolution of the image.

A second approach is to use a higher-resolution display. For example, changing from 512 × 512 to 1,024 × 1,024 resolution will improve the image. However, such an approach is very expensive, it only reduces the aliasing error by a factor of two, and it still leaves the aliasing artifacts quite visible.

Supersampling is a third antialiasing technique, in which the display is treated as if it has a higher resolution than is actually the case. Each pixel on the display is composed of several higher-resolution pixels or *subpixels*. The values of the subpixels for this virtual higher-resolution display are computed in the manner described above, each subpixel being assigned a value of *full on* or *full off*. Then the values of the set of subpixels that constitute each pixel of the actual display are averaged according to some rule, the result being a display pixel that may be set to a gray level between full on and full off. Thus the use of this technique requires the ability to set the intensity of individual pixels to intermediate values between full on and full off. This technique increases the time required to draw a line. Increasing the virtual resolution of the display by a factor of n requires n^2 subpixels per pixel, with a computational load also increasing by a factor of n^2, which is unacceptably high.

The fourth and final technique consists of prefiltering the data before sampling. As is described in the next section, this method takes into account the reasons for the aliasing artifacts and takes steps to remove them. Most antialiasing methods in common use employ prefiltering. In particular, the common implementation of the antialiased paintbrush is filtered to smooth its edges.

As with the supersampling technique, prefiltering techniques require the ability to set the intensity level of any pixel to intermediate values between full on and full off. A consequence of this requirement is that the framebuffer must have more than one bit to represent the gray level or color of each pixel.

A further discussion of the aliasing problem and of approaches to antialiasing can be found in Rogers (1985) and Foley and Van Dam (1981).

Antialiased Lines

The term *aliasing* is used to describe the result of a signal being sampled at a rate lower than twice its frequency, as illustrated in Figure 2. The signal at the top of Figure 2 is sampled at a frequency indicated by the dots on the waveform. The result of this sampling (Figure 2, bottom) is a signal that is an *alias* of the original and bears little obvious relation to the original. The line-drawing algorithm used to draw the lines in Figure 1 can be thought of as taking a sample at the center point of each pixel.

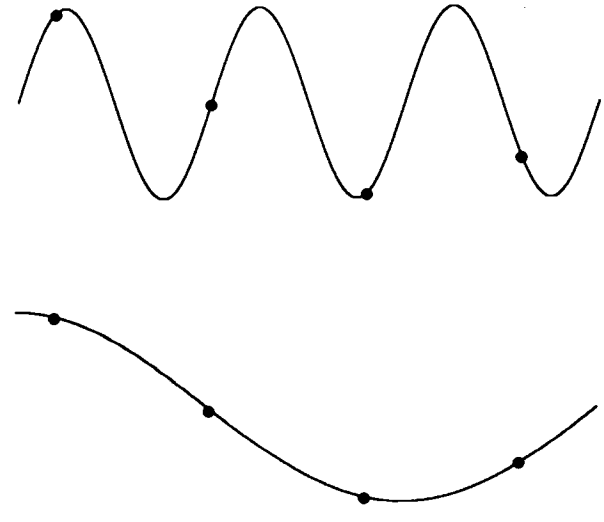

Figure 2. If the sample frequency of a signal (top) is less than twice that of the sampled frequency, the result shows a new frequency or alias of the original frequency (bottom).

If the line (modeled as a polygon or band with a width of one pixel) crosses this center point, the pixel is turned on, if the line does not cross the point, the pixel is off.

Prefilter antialiasing algorithms operate by setting pixel values according to a blending function

$$pixel(i,j) = \alpha I + (1-\alpha) I_{back},$$

where I is the color of the object being drawn, I_{back} is the previous color of the pixel, and α is the result of a low-pass filter applied to the object at (i,j). In general $0 \leq \alpha \leq 1$.

There are many methods to calculate α, which differ in how they trade off computational accuracy against time required for the calculations. Many implementations use an unweighted filter, one that is constant throughout the convolution mask. This results in α being equal to that

```
#define INTENSITY 15
#define abs(x) (( x < 0 ) ? (-(x)) : x )
#define BETWEEN(x1,mid,x2)\
 (x1>=mid && x2<=mid || x1<=mid && x2>=mid ? 1 : 0)

Draw_Antialiased_Line(x1, y1, x2, y2)
int x1, y1, x2, y2;
{ int x, y, dx, abs_dx, dy, abs_dy, dx2, dy2, A, Aerr, Tri;
  int INTENSITYlessTri, Sq, Sqerr, over, under, r, q, L, incr;

   if( x1==x2 && y1==y2 ) return(0);

   abs_dx = ( abs( dx = x2 - x1) );
   abs_dy = ( abs( dy = y2 - y1) );

   if( (abs_dx >= abs_dy && dy < 0 )
      || ( abs_dx < abs_dy && dx < 0) ) {
         swap( &x1, &x2 );
         swap( &y1, &y2 );
         dx = -dx;
         dy = -dy;
   }

   if( abs_dx >= abs_dy ) {
      incr =( ( dx > 0 ) ? 1 : -1 );
      dx2 = abs_dx * 2;
      A = (abs_dx + INTENSITY * (abs_dx + abs_dy) ) / dx2;
      Aerr = (abs_dx + INTENSITY * (abs_dx + abs_dy)) % dx2;
      Tri = (abs_dx + INTENSITY * abs_dy) / dx2;
      INTENSITYlessTri = INTENSITY - Tri;
      Sq = (2 * INTENSITY * abs_dy) / dx2;
      Sqerr = (2 * INTENSITY * abs_dy) % dx2;
      over = 2 * (abs_dy - abs_dx);
      under = 2 * abs_dy;
      r = 2 * abs_dy - abs_dx;
      q = y1;

      for( x = x1; BETWEEN( x1, x, x2); x += incr ) {
         if( r >= 0 ) {
            A -= INTENSITY;
            L = (Tri - A)>>1;
            pixel( x, q + 1, L + A );
            pixel( x, q, INTENSITYlessTri );
            pixel( x, q - 1, L );
            r += over;
            q++;
         } else {
            pixel( x, q, A );
            pixel( x, q - 1, INTENSITY - A );
            r += under;
         }
         A += Sq;
         Aerr += Sqerr;
         if( Aerr >= dx2 ) {
            A++;
            Aerr -= dx2;
         }
      }
   }
   else {
      incr = ( ( dy > 0 ) ? 1 : -1 );
      dy2 = abs_dy * 2;
      A = (abs_dy + INTENSITY * (abs_dx + abs_dy) ) / dy2;
      Aerr = (abs_dy + INTENSITY * (abs_dx + abs_dy)) % dy2;
      Tri = (abs_dy + INTENSITY * abs_dx) / dy2;
      INTENSITYlessTri = INTENSITY - Tri;
      Sq = (2 * INTENSITY * abs_dx) / dy2;
      Sqerr = (2 * INTENSITY * abs_dx) % dy2;
      over = 2 * (abs_dx - abs_dy);
      under = 2 * abs_dx;
      r = 2 * abs_dx - abs_dy;
      q = x1;

      for( y = y1; BETWEEN( y1, y, y2); y += incr ) {
         if( r >= 0 ) {
            A -= INTENSITY;
            L = (Tri - A)>>1;
            pixel( q + 1, y, L + A );
            pixel( q, y, INTENSITYlessTri );
            pixel( q - 1, y, L );
            r += over;
            q++;
         } else {
            pixel( q, y, A );
            pixel( q - 1, y, INTENSITY - A );
            r += under;
         }
         A += Sq;
         Aerr += Sqerr;
         if( Aerr >= dy2 ) {
            A++;
            Aerr -= dy2;
      }}}}

swap( x, y )
int *x, *y;
{ int t;
   t = *x;  *x = *y;  *y = t;
}

pixel( x, y, intensity )
int x, y, intensity;
{ extern struct RastPort  *rp;
   SetAPen( rp, intensity );
   WritePixel( rp, x, y );
}
```

Figure 3. "C" source code for antialiased lines.

fraction of the area around the pixel that is covered by the line (Crow, 1978). Others model the pixel using a Gaussian function, so that the area of a line closer to the center of the pixel contributes more to the value of α than does an equal area further from the center (Feibush, Levoy, & Cook, 1980). Although this more closely models the actual physics of the monitor, it is much more time-consuming than the use of constant filters.

In experiments conducted with graphics on a personal computer, the tradeoff between image fidelity on one hand and speed of calculation and ease of implementation of the other normally favors the more practical and simpler antialiasing techniques. To the authors' knowledge, there have been no studies comparing the differences between the results of various antialiasing techniques in terms of the viewer's perception of quality or the viewer's ability to comprehend the image. These differences are apparently much less than the differences between images with or without antialiasing.

Siding with ease of implementation and speed of calculation, we describe a very rapid antialiased-line-drawing technique based on an algorithm by Field (1984). Field actually presents two algorithms for drawing antialiased lines, both of which compute αI directly without lookup tables or floating-point operations. Inner loop multiplications are avoided completely. The first algorithm, the one we use, employs approximation techniques to compute the filter response values. Figure 3 shows this algorithm coded as a C program (Kernighan & Ritchie, 1978) for the Commodore Amiga. The Appendix shows a sample program that calls the routine shown in Figure 3. Figure 4 presents an image of the same lines as those in Figure 1, but rendered with the use of this algorithm.

This algorithm affects pixels in a parallelogram, with a major axis parallel to the line being drawn and a vertical minor axis if $|\Delta x| < |\Delta y|$; otherwise the minor axis is horizontal. The pixels about the point where adjoining line segments meet are normally handled in a consistent and continuous manner. Indeed, drawing a line from (x,y) to $(x+n,y+m)$ would give the same result as drawing two line segments, one from (x,y) to $[x+(n/k),y+(m/k)]$ and the second from $[x+(n/k),y+(m/k)]$ to $(x+n,y+m)$, assuming that both n and m are integers divisible by k. However, if $|\Delta x| < |\Delta y|$ for one line segment, and not for the abutting segment, the pixels on the exterior angle will not be properly shaded. Extending the algorithm to handle this case would both increase its complexity and decrease its speed considerably.

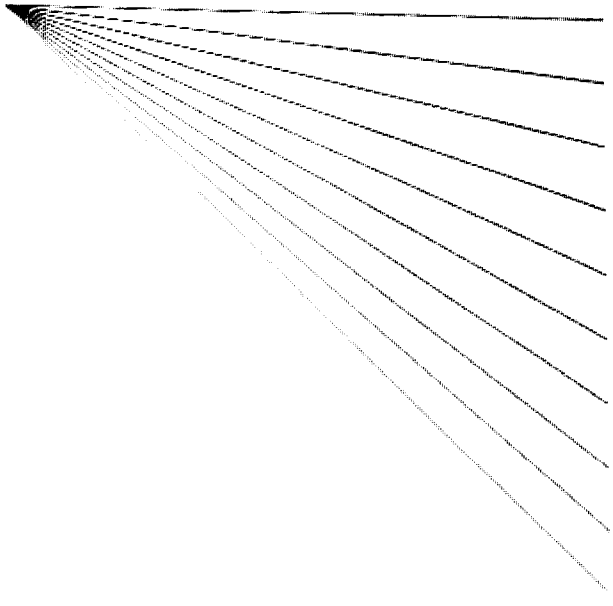

Figure 4. Lines drawn by the antialiased line drawing code listed in Figure 3.

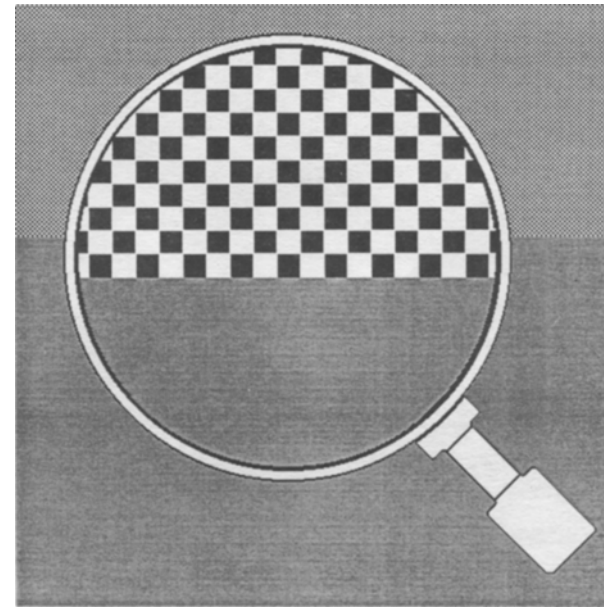

Figure 5. Technique used to compare the actual intensities of different pixel values on a display.

The algorithm assumes that the color of the background is black. White lines drawn on a black screen present a problem only when the lines cross each other, in which case a small portion of the line that has been drawn first will be made darker. Again, extending the algorithm to correct for this situation, while very simple, would significantly decrease its speed.

Gamma Correction

Lines produced on a graphics display may suffer from a problem in addition to aliasing. Even after antialiasing has been applied, lines that are not perfectly horizontal or vertical may have a striped look, often called the *barber pole effect*. Furthermore, the stripes move along the line when it is rotated. The lines in Figure 4 suffer from this problem.

The barber pole effect can be reduced by the use of *gamma correction*. Gamma correction takes into account the nonlinear function relating the numbers in the gray scale used to specify an intensity on the screen and the resulting amount of light emitted from the screen phos-

phor. Usually, the gray-scale values are linearly related to the voltage sent to the control circuitry of the CRT. However, there is a nonlinear relationship between this voltage and the resulting light emission, which is shown in the following expression:

$$L = kV^{\gamma}$$

where V is the voltage, k is a constant, L is the emitted light and γ is a constant that depends on the monitor used, as well as on the brightness and contrast settings of the monitor. Because the values that are placed in each framebuffer pixel are used to control the output voltage, not the light, if one has 16 gray levels represented by pixel values between 0 and 15 inclusive, it is quite likely that half intensity may be represented by a pixel value some distance from 7 or 8.[1]

Fortunately, discovering the value of γ and then performing gamma correction is not difficult. Note, however, that the value of γ will vary from monitor to monitor and, on any single monitor, from one day to the next (because of inconsistencies in voltage and temperature). In practice, the amount of acceptable variability will depend on the nature of the experimentation. Careful psychophysical work may require that the value of γ be estimated frequently.

To discover γ for a particular monitor at a particular time, one may place two rectangles on the screen beside each other. One rectangle should have a pattern of pixels with half of them at full intensity and the other half at zero intensity. All the pixels of the second rectangle should be at the same intensity, approximately in the middle of the lightness range as illustrated in Figure 5. The viewer is then given control over the intensity value of the pixels of the second rectangle and is asked to set its intensity such that its brightness matches that of the first rectangle. This is easily done by modifying the hardware color lookup table.[2] The pixel value required in the second rectangle to make this brightness match is recorded, and considered to be half intensity.

At this stage, the value of γ can be computed directly. Alternatively, to ensure the accuracy of the computation of γ, the process can be repeated after changing the zero-intensity pixels in the first rectangle to the newly found half-intensity values so that the three-quarter intensity value can be found. This procedure can be repeated several times until enough points are available to produce a graph that relates pixel values and screen intensities. A lookup table relating the pixel values with the screen intensities can be filled from interpolated points on the graph. Alternatively, the gamma can be computed, and the table entries then calculated. Table 1 was filled by means of the second technique (the computed gamma was 2.0; gammas for most monitors range from 1.8 to 2.8), but a table derived directly from a hand-drawn graph differed only a little from the computed graph. Note that in the table, the range of computed gray-scale values has been increased to 64. If it had been left at 16 values (the number of possible intensities on the display), certain intensities could never be displayed, because of the nonlinearity of the mapping. This would result in a slight reduction in the accuracy of the intensities of the darker pixels.

With a software lookup table loaded so that pixel values correspond directly with the pixel brightness (see the code in Figure 6) the lines previously displayed in Figures 1 and 4 have been replotted in Figure 7. When displayed on a monitor, the gamma-corrected images appear different in two ways. First, the barber pole effect is reduced in Figure 7, an effect particularly apparent when lines are in motion. Second, the intensity of the lines varies less if they are rotated. In general they appear to be darker than the uncorrected version. These differences may not be noticeable on the printed page, but they are certainly apparent on the monitor, particularly for lines that are in

Table 1
Lookup Table to Map Computed Intensity Values (LUT Index) to Closest Pixel Intensities (Computed Using γ = 2.0)

LUT Index	LUT Value	LUT Index	LUT Value	LUT Index	LUT Value	LUT Index	LUT Value
0	0	16	8	32	11	48	13
1	2	17	8	33	11	49	14
2	2	18	8	34	11	50	14
3	3	19	8	35	11	51	14
4	4	20	9	36	12	52	14
5	4	21	9	37	12	53	14
6	4	22	9	38	12	54	14
7	5	23	9	39	12	55	14
8	5	24	9	40	12	56	15
9	6	25	10	41	12	57	15
10	6	26	10	42	13	58	15
11	6	27	10	43	13	59	15
12	6	28	10	44	13	60	15
13	7	29	10	45	13	61	15
14	7	30	11	46	13	62	15
15	7	31	11	47	13	63	15

```
int Gamma[64] =        /* Gamma correction look up table*/
    {0, 2, 2, 3, 4, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7, 7,
     8, 8, 8, 8, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10,
     11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12,
     13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14,
     14, 14, 15, 15, 15, 15, 15, 15, 15, 15};

#define INTENSITY 63
#define abs(x) (( x < 0 ) ? (-(x)) : x )
#define BETWEEN(x1,mid,x2) \
 (x1>=mid && x2<=mid || x1<=mid && x2>=mid ? 1 : 0)

Draw_Gamma_Corrected_Line(x1, y1, x2, y2)
int x1, y1, x2, y2;
{ int x, y, dx, abs_dx, dy, abs_dy, dx2, dy2, A, Aerr, Tri;
  int INTENSITYlessTri, Sq, Sqerr, over, under, r, q, L, incr;

   if( x1==x2 && y1==y2 ) return(0);

   abs_dx = ( abs( dx = x2 - x1) );
   abs_dy = ( abs( dy = y2 - y1) );

   if( (abs_dx >= abs_dy && dy < 0 )
      || ( abs_dx < abs_dy && dx < 0) ) {
         swap( &x1, &x2 );
         swap( &y1, &y2 );
         dx = -dx;
         dy = -dy;
   }

   if( abs_dx >= abs_dy ) {
      incr =( ( dx > 0 ) ? 1 : -1 );
      dx2 = abs_dx * 2;
      A = (abs_dx + INTENSITY * (abs_dx + abs_dy) ) / dx2;
      Aerr = (abs_dx + INTENSITY * (abs_dx + abs_dy)) % dx2;
      Tri = (abs_dx + INTENSITY * abs_dy) / dx2;
      INTENSITYlessTri = INTENSITY - Tri;
      Sq = (2 * INTENSITY * abs_dy) / dx2;
      Sqerr = (2 * INTENSITY * abs_dy) % dx2;
      over = 2 * (abs_dy - abs_dx);
      under = 2 * abs_dy;
      r = 2 * abs_dy - abs_dx;
      q = y1;

      for( x = x1; BETWEEN( x1, x, x2); x += incr ) {
         if( r >= 0 ) {
            A -= INTENSITY;
            L = (Tri - A)>>1;
            pixel( x, q + 1, Gamma[ L + A ] );
            pixel( x, q, Gamma[ INTENSITYlessTri ] );
            pixel( x, q - 1, Gamma[ L ] );
            r += over;
            q++;
         } else {
            pixel( x, q, Gamma[ A ] );
            pixel( x, q - 1, Gamma[ INTENSITY - A ] );
            r += under;
         }
         A += Sq;
         Aerr += Sqerr;
         if( Aerr >= dx2 ) {
            A++;
            Aerr -= dx2;
         }
      }
   }
   else {
      incr = ( ( dy > 0 ) ? 1 : -1 );
      dy2 = abs_dy * 2;
      A = (abs_dy + INTENSITY * (abs_dx + abs_dy) ) / dy2;
      Aerr = (abs_dy + INTENSITY * (abs_dx + abs_dy)) % dy2;
      Tri = (abs_dy + INTENSITY * abs_dx) / dy2;
      INTENSITYlessTri = INTENSITY - Tri;
      Sq = (2 * INTENSITY * abs_dx) / dy2;
      Sqerr = (2 * INTENSITY * abs_dx) % dy2;
      over = 2 * (abs_dx - abs_dy);
      under = 2 * abs_dx;
      r = 2 * abs_dx - abs_dy;
      q = x1;

      for( y = y1; BETWEEN( y1, y, y2); y += incr ) {
         if( r >= 0 ) {
            A -= INTENSITY;
            L = (Tri - A)>>1;
            pixel( q + 1, y, Gamma[ L + A ] );
            pixel( q, y, Gamma[ INTENSITYlessTri ] );
            pixel( q - 1, y, Gamma[ L ] );
            r += over;
            q++;
         } else {
            pixel( q, y, Gamma[ A ] );
            pixel( q - 1, y, Gamma[ INTENSITY - A ] );
            r += under;
         }
         A += Sq;
         Aerr += Sqerr;
         if( Aerr >= dy2 ) {
            A++;
            Aerr -= dy2;
         }
      }
   }
}

swap( x, y )
int *x, *y;
{ int t;
  t = *x;  *x = *y;  *y = t;
}

pixel( x, y, intensity )
int x, y, intensity;
{ extern struct RastPort  *rp;
  SetAPen( rp, intensity );
  WritePixel( rp, x, y );
}
```

Figure 6. "C" source code for antialiased lines with gamma correction.

motion. The technique suggested here for gamma correction requires only a single setup operation. Once derived, the use of gamma-corrected values does not require any additional computation during the rendering of the image on the screen.

Extensions

The antialiasing algorithm discussed in this paper is limited to the drawing of lines on a black background. Many techniques exist that perform the antialiasing of polygon edges (Feibush, Levoy, & Cook, 1980), anti-

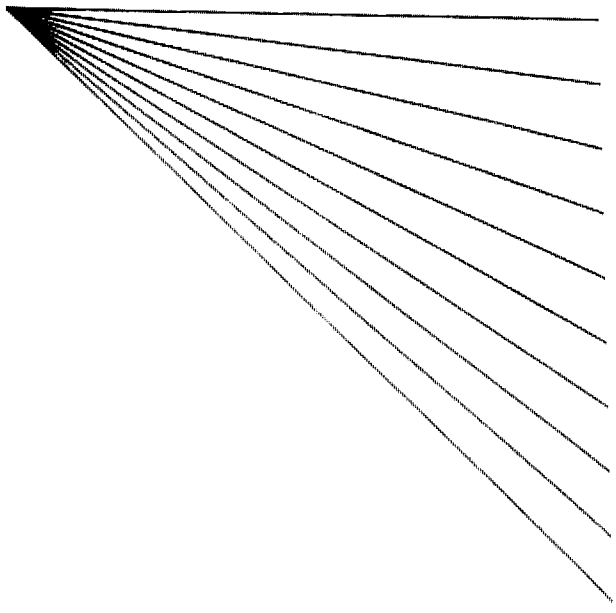

Figure 7. Lines drawn by the antialiased line code with gamma correction listed in Figure 6.

aliasing between arbitrary colors (as opposed to assuming a black background as in this paper (Turkowski, 1986), and antialiasing for the very realistic images created by *ray tracing* (Mitchell, 1987).

Just as this paper shows how to find the gamma of a monitor, Cowan (1983) describes a technique to find the functions required for mapping monitor gun voltages into CIE coordinates (XYZ color space). In addition, Catmull (1979) discusses the use of lookup tables for film recording, taking into account both the gamma correction required on the monitor, and the nonlinearities in the response of the film.

Hardware and Software Considerations

The computer routines displayed in this article were written in C for the Commodore Amiga. The display was in high-resolution mode, which results in four bits being available for each pixel, and hence 16 possible gray levels. The programs compile and run using either Lattice C or Manx C.

Even with the relatively efficient antialiasing technique outlined above, the time needed to render a given length of line is considerably longer than the time needed to draw a line that has not been antialiased. The difference in drawing time is especially large in computer systems that have hardware-based line-drawing capabilities (such as the Amiga), because the computations required for the antialiased line are all performed in software. Nonetheless, we have found that relatively complex stimuli (consisting of up to 300 short line segments) can be rendered on the Amiga in less than 5 sec. Thus, these stimuli can be "painted" on an invisible screen during the intertrial interval. Of course, the feasibility of the antialiasing technique will depend on a number of factors, such as the speed of the computer, the complexity of the images to be rendered, and the amount of time available for the rendering process.

We know of no other research that has addressed the question of how many different levels of gray, or, alternatively, how many bits per pixel, are required to produce "reasonable" antialiased lines. From our experience with the Amiga, however, it is clear that four bits are sufficient to render jaggies (that are quite apparent when lines are rendered without antialiasing) virtually imperceptible at normal viewing distances. Nor has any research assigned comparative measures to the quality of lines produced by different antialiasing techniques. Ultimately, it would be useful to carry out a series of experiments similar to those of Booth et al. (1987), in order to map out how performance (and/or appearance) trades off with the number of bits per pixel used to render the antialiased lines.

REFERENCES

BOOTH, K. S., BRYDEN, M. P., COWAN, W. B., MORGAN, M. F., & PLANTE, B. L. (1987). On the parameters of human visual performance: An investigation of the benefits of antialiasing. *IEEE Computer Graphics & Applications*, **7**, 34-41.

BRESENHAM, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*, **4**(1), 25-30.

CATMULL, E. (1979). A tutorial on compensation tables. *Computer Graphics*, **13**(2), 1-7.

COWAN, W. B. (1983). An inexpensive scheme for calibration of a colour monitor in terms of CIE standard coordinates. *Computer Graphics*, **17**(3), 315-321.

CROW, F. C. (1977). The aliasing problem in computer-synthesized computer generated images. *Communications ACM*, **20**(11).

CROW, F. C. (1978). The use of grayscale for improved raster display of vectors and characters. *Computer Graphics*, **12**(3), 1-5.

FEIBUSH, E. A., LEVOY, M., & COOK, R. L. (1980). Synthetic texturing using digital filters. *Computer Graphics*, **14**(3), 294-301.

FIELD, D. (1984). Two algorithms for drawing anti-aliased lines. In *Proceedings, Graphics Interface '84* (pp. 87-95). Toronto: The Canadian Information Processing Society.

FOLEY, J. D., & VAN DAM, A. (1981). *Fundamentals of interactive computer graphics*. Reading, MA: Addison-Wesley.

KERNIGHAN, B. W., & RITCHIE, D. M. (1978). *The C programming language*. Englewood Cliffs, NJ: Prentice-Hall.

LINDHOLM, J. M. (1988, November). *Temporal factors affecting perception of raster graphics images*. Paper presented at the annual meeting of the Psychonomic Society, Chicago, IL.

MITCHELL, D. P. (1987). Generating antialiased images at low sampling densities. *Computer Graphics*, **21**(4), 65-69.

ROGERS, D. F. (1985). *Procedural elements for computer graphics*. New York: McGraw-Hill.

TURKOWSKI, K. (1986). Anti-aliasing in topological color spaces. *Computer Graphics*, **20**(4), 307-314.

NOTES

1. Monitors are deliberately designed to have a gamma function to match the response of the visual system. Thus, the voltage and (perceived) brightness are linearly related. However, as we are combining the intensity of several pixels to form the impression of an image, we must control the actual pixel intensity, not its perceived brightness.

2. The hardware color lookup table is used to map pixel values into specific gray levels. Setting Entry 0 of the lookup table to full intensity will cause all pixels with a value of zero to be displayed at full intensity. In our situation, Entry 0 is set to zero intensity, the maximum level (Level 15) is set to full intensity, and the intermediate levels are set according to the gamma function.

APPENDIX

A sample program that calls the routine shown in Figure 3 to draw antialiased lines. This program could also be used to call the routine shown in Figure 6 by changing "Draw_Antialiased_Line" to "Draw_Gamma_Corrected_Line."

```
#include <stdio.h>
#include <clib/macros.h>
#include <intuition/intuition.h>

struct Library *GfxBase, *IntuitionBase = NULL;
struct Screen  *scr = NULL;
struct ViewPort *vp = NULL;
struct RastPort *rp = NULL;
struct NewScreen ns = {
  0,0,640,400,4,1,0,HIRES|LACE,CUSTOMSCREEN,NULL,NULL,NULL,NULL };
unsigned short colours[] = {
  0x0000, 0x0111, 0x0222, 0x0333, 0x0444, 0x0555, 0x0666, 0x0777,
  0x0888, 0x0999, 0x0aaa, 0x0bbb, 0x0ccc, 0x0ddd, 0x0eee, 0x0fff };
main() {
  GfxBase = OpenLibrary( "graphics.library",0 );
   if ( GfxBase == NULL ) Zap( "Can't open GfxLib" );
  IntuitionBase = OpenLibrary( "intuition.library",0 );
   if ( IntuitionBase == NULL ) Zap( "Can't open Intuition" );
  scr = OpenScreen( &ns );
   if ( scr == NULL ) Zap( "Can't open screen" );
  rp = &( scr->RastPort );
  vp = &( scr->ViewPort );
  LoadRGB4( vp, &colours[0], 16L );
  SetDrMd( rp, JAM1 );
  Draw_Antialiased_Line( 320L, 100L, 370L, 200L ); /* draw diamond */
  Draw_Antialiased_Line( 370L, 200L, 320L, 300L ); /* using       */
  Draw_Antialiased_Line( 320L, 300L, 270L, 200L ); /* antialiased */
  Draw_Antialiased_Line( 270L, 200L, 320L, 100L ); /* lines       */
  printf( "Press 'q' to quit: " );
  while ( getchar() != 'q' );
  Zap( "Done" );
}

Zap( text )
char *text; {
  if ( scr ) CloseScreen( scr );
  if ( IntuitionBase ) CloseLibrary( IntuitionBase );
  if ( GfxBase ) CloseLibrary( GfxBase );
  printf( " %s\n", text );
  exit( 0 );
}
```

(Manuscript received April 22, 1988;
revision accepted for publication December 14, 1988.)