An Algorithm for Incremental Anti-Aliased Lines and Curves

by

Roger Ebbe-Jan Hagen

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Electrical Engineering

at the Massachusetts Institute of Technology

January 1988

© Roger E. Hagen, 1988

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

January 25, 1988

Certified by

William F. Schreiber
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by

Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses
An Algorithm for Incremental Anti-Aliased Lines and Curves

by

Roger Ebbe-Jan Hagen

Submitted to the
Department of Electrical Engineering and Computer Science
on 25 January 1987 in partial fulfillment of the requirements
for the degree of Bachelor of Science

Abstract

Rendering of lines and curves on raster displays produce "jagged" edges which are the result of aliasing. Methods for removal of these aliasing artifacts involve increased sampling and low pass filtering which have the effect of smoothing the edge. The high computational cost of these methods of anti-aliasing has precluded any wide-spread use of these methods.

This paper presents an algorithm to incrementally render anti-aliased lines, conics, and cubics at low time and computational cost. The algorithm yields exact results for lines and very good results for second and third degree curves.

Thesis Supervisor: William F. Schreiber
Title: Professor of Electrical Engineering and Computer Science
Contents

1 Introduction .................................................. 1

2 The Problem .................................................. 3
2.1 Two-Dimensional Sampling ................................. 3
2.2 Spatial Filtering ............................................. 4
2.3 Previous Efforts ............................................. 7

3 The Algorithm .................................................. 11
3.1 Filter Table Construction ................................. 11
3.2 Distance Metrics ........................................... 14
3.3 Anti-Aliasing a Straight Line ............................. 15
3.4 Anti-Aliasing a Quadrilateral ............................. 16

4 Higher Order Curves .......................................... 18
4.1 Equations and Expansions ................................. 18
4.1.1 Conic Function ........................................... 19
4.1.2 Cubic Function ........................................... 21
4.2 Getting the Normal ......................................... 23

5 Implementation ............................................... 26

6 Limitations and Future Work ............................... 28
6.1 What Does Not Work ........................................ 28
6.2 Using the Radius of Curvature ............................ 29
6.3 Width Scans ............................................... 31

7 Conclusion ..................................................... 34
8 Results

9 Appendix: Code Listing
9.1 Setting Up The Table ........................................ 43
9.2 Line Algorithm .................................................... 45
Acknowledgments

I would like to thank a great number of people, and since this is single spaced I might as well do a fair number of them. Special thanks to mom who made it possible for me to pursue fun endeavors (such as MIT, ha ha) and for being a great anarchical force in this part of the galaxy; to Lydia for keeping my sanity in a safe place till I need it again; to my sister for no apparent reason; to Samurai Cat for being a dude; to George and Max for their helpful distractions; to Don Hatfield, Hubert Delany, Charles Forsythe, Doug Church, Tareq Hoque and Bob Mollitor who provided ideas, insight, bitdidling, and support; to Ted Young for getting the picture; to Matt Baker, Ed Demko, Chuck Menoche, Nick Lucia, Russell Courter, Ralph Acquaro, and John Lamar for their friendship; to William F. Schreiber for his help in outlining what I should be doing; and finally, to all the makers of fine caffeine everywhere, especially Pepsi-Cola for Pepsi and Mountain Dew, Coca-Cola for Coke and Dr. Pepper, and Morning Thunder Tea for that extra caffeine in the morning.
Chapter 1

Introduction

The flexibility of the raster display in depicting realistic scenes makes it desirable to remove the degradation common to the computer screen, aliasing artifacts. This characteristic is exhibited by the stair-step pattern of lines and edges that do not happen to align with the coordinate axes. Any continuous image will exhibit aliasing after sampling if the original image has component frequencies above \( \frac{1}{2} \) the sample rate. Smoothing edges enhances the perceived image quality and the apparent resolution.

Present methods for anti-aliasing involve either convolution with a low pass filter or increasing the sample rate. Both schemes involve considerable cost; sample rate increase involves a great deal of complexity, and filtering requires expensive time and computation. Algorithms have been derived to render simple shapes, such as lines and polygons, by using incremental methods. The test of these methods is their ability to produce high quality, quickly rendered images. This thesis will present an efficient algorithm for arbitrary lines and convex quadrilaterals, as well as second and third
degree curves. It requires only adds, compares and table lookups for lines and quadrilaterals, while the higher order curves use a small amount of division for each scanline. The application of the algorithm is to enhance the resolution of interactive graphics without sacrificing response time.
Chapter 2

The Problem

2.1 Two-Dimensional Sampling

Given a two-dimensional continuous function \( f(x, y) \), the sampling process for a point \((x_0, y_0)\) is:

\[
\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \delta(x - x_0, y - y_0) \, dx \, dy = f(x_0, y_0)
\]

A two dimensional sampling function consists of a train of impulses separated by \( \Delta x \) in the \( x \) direction and \( \Delta y \) units in the \( y \) direction. A sampled function is obtained by forming the product \( f(x, y)s(x, y) \) where \( s(x, y) \) is the impulse function. In the frequency domain, the equivalent operation is the convolution of \( S(u, v) \) and \( F(u, v) \) where \( S \) and \( F \) are the Fourier transforms of \( s \) and \( f \). \( S(u, v) \) is a train of impulses with separation \( \frac{1}{\Delta x} \) and \( \frac{1}{\Delta y} \) in the \( u \) and \( v \) directions.

If \( 2W_u \) and \( 2W_v \) are the widths in the \( u \) and \( v \) directions of a rectangular
region $R$, and if $\frac{1}{\Delta x} > 2W_u$ and $\frac{1}{\Delta y} > 2W_v$, then one of the periods may be recovered if the convolution of $S(u, v) \ast F(u, v)$ is multiplied by

$$G(u, v) = \begin{cases} 
1 & (u, v) \text{ inside one of the rectangles enclosing } R \\
0 & \text{ elsewhere}
\end{cases}$$

The function $f(x, y)$ can be recovered by the inverse Fourier transform of $G(u, v)[S(u, v) \ast F(u, v)]$. Thus, the two-dimensional sampling theorem states that a band-limited function $f(x, y)$ can be recovered from samples where

$$\Delta x \leq \frac{1}{2W_u}$$

$$\Delta y \leq \frac{1}{2W_v}$$

[Gonz87]

2.2 Spatial Filtering

Aliasing artifacts are caused by sampling objects with sharp edges. In the Fourier spectrum, any signal with a sharp edge is composed of frequency components which include infinite frequency. Raster screens can not represent frequency components higher that half the sample rate, known as the Nyquist rate. Higher frequencies are aliased into looking like low frequencies. These problems characteristically occur in three specific situations: (1) along edges of images, (2) in small objects, and (3) in regions with com-
Frequency-domain representation of a sampled two-dimensional, band-limited function.

Figure 2.1: Sampling

Complicated detail. To ensure that no frequencies in the signal exceed one-half the sampling frequency, it is necessary to filter out the higher frequencies. The image must be convolved with a two-dimensional filter. Fast convolution methods include the fast Fourier transform which requires an excessive amount of computation, and direct convolution which is more viable than the FFT if the convolution kernel is small in relation to the image array [Crow78].

According to William Leler, a 17 inch diagonal measure monitor viewed at 25 inches would have to have a resolution of 3577 lines to eliminate most of the effects of aliasing [Lel80]. Even at such a high resolution some aliasing effects could still be perceived. With present high quality displays in the 1024×1024 range, anti-aliasing is desirable using low pass filtering [Crow77]. This results in a slight blurring of the image, which gives a higher apparent
Samples are the same in both signals.

Figure 2.2: Aliasing

A line and its Fourier spectrum.

Figure 2.3: Example of Infinite Frequency
resolution. The pixels on the screen are point sources and the eye cannot
distinguish between size and brightness[Lei80].

2.3 Previous Efforts

Direct filtering would require the convolution of some \( l \times m \) filter array with
an arbitrary \( k \times n \) picture array which is usually very expensive, dependent
on the values of \( k, l, m, \) and \( n \). To avoid the costs incurred by such direct
methods, much research has been done to either reduce the cost of filtering
or to use alternative methods to make the problem less noticeable.

Methods of making aliasing less noticeable include nonuniform sampling
and optimized sampling. Nonuniform sampling does not eliminate aliasing,
but changes the characteristics of aliasing to make it less noticeable[Reeves87].
A large neighborhood of low-density base samples are used to detect high
frequency, and supersampled if necessary[Mitch87]. Optimized sampling in-
volves random sampling with a statistical test to determine whether enough
samples have been used for that portion of the image[Lee85].

Methods have been derived that achieve the convolution by integration
of the filter function over the region of interest. However, integration is also
a costly method of filtering. There has been a gradual evolution in mak-
ing use of integration using incremental techniques for value determination
directly or by tabular methods.

Pitteway and Watkinson adapted Bresenham's algorithm for line ren-
dering into an incremental anti-aliasing line algorithm. For a line with
a slope in the first octant, the variable \( d \) is associated with the equation
of a line: \( f(x, y) = mx + c - y \). The equation is evaluated at the point \((i + 1, j + \frac{1}{2})\), where the present point selected would be \((i, j)\). If \(d\) is negative, the point \((i + 1, j + \frac{1}{2})\) is above the line and a step in \(x\) is required. If \(d\) is positive, a diagonal move is required. If the variable \(d\) is used in split integer format and operates in the region \(0 \leq d < 1\), then \(d\) can be used to select a grey shade that corresponds to a pixel value that is equal to a filter integration over the line[ Pitt80].

Gupta and Sproull expanded on the previous method to consider variable width lines, polygons, and the endpoints for lines. Using a radially symmetric filter with an enclosing volume of 1, a pixel's intensity is proportional to the volume of the filter intersected by the line. Because of symmetry, the parameters needed to determine the volume intersected are the thickness of the line and the perpendicular distance from the line center. A table is constructed by convolving the filter function and the line intensity by numerically integrating the filter function over the region covered by the line.

Their algorithm used a one unit thick line in the first octant, with obvious extrapolation to other octants. Using a form of Bresenham's algorithm to track the center line, three pixels were shaded which would affect at most three pixels in each column since a 1 unit radius filter would only effect two or three pixels in each column (i.e. at \((x, y), (x, y - 1),\) and \((x, y + 1)\)). If \(v\) is the vertical distance form a line with slope \(m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}\), then perpendicular distances are related by a factor of \(c = \frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}\), such that the distance is equal to \(cv\). The distance value was used to read the table.

Since the algorithm is implemented using split integer format and using
adds, compares, and table lookups, it is very efficient. For endpoints of a line, tables are constructed based on slopes from 0 to 1 in intervals of \( \frac{1}{16} \) and using mirroring and transposition transformations to do the other octants\[Gup81]\.

The use of coordinate transformation in anti-aliasing was applied by Kenneth Turkowski to anti-alias lines and polygons. The principle was the same as with Gupta and Sproull in finding pixel values dependent on point-line or point-segment distance, with the object to reduce the problem of finding the distance in two dimensions to that of finding the distance in one dimension. Turkowski used a fast CORDIC rotation algorithm to transform a line segment onto an \( x' \) axis with the point-segment distance being simply the \( y' \) coordinate\[Turk82]\.

Fujimoto and Iwata derived an algorithm which again utilized point line distance, but did not involve table lookup. They used a method where the point-segment distance gave the pixel value directly and used that value to determine the rendering of the line. They limited their method to lines restricted to a maximum of three pixels shaded in each column for lines with slopes in the first quadrant. The method is extremely fast but does not consider end points\[Fuji83]\.

Working along similar lines of point-line, point-segment distances, Hubert Delany derived an orientation independent method for lines with endpoint filtering. He utilized the same distance descriptors as Gupta and Sproull, but used them to keep track of several lines that made up a rectangular object that represented a line. A table was constructed using the integration of the filter function over a corner. The filter value was read
using the distance values given by the distance descriptors to the lines. Using appropriate addition and subtraction of filter fragments, a pixel value was derived that gave the filtered value for endpoints and was orientation independent [Del86].
Chapter 3

The Algorithm

3.1 Filter Table Construction

Straight lines are filtered by treating the line as a rectangle with four edges (See Figure 3.1). A filter table is constructed based on the area covered by two intersecting edges and integrating the filter function measured by two distance indexes that gave a perpendicular distance to the respective edges. The value of the integration is the pixel value at that distance from the two edges (See Figure 3.2). The radius of the filter was divided into \( \frac{1}{16} \) so that distance indexes go from 0 to 31 with 15 being right on an edge. A distance index of 31 is used when the center of the filter function is a radius distance or more inside the edge, and a distance index of 0 is used when the center of the filter function is a radius distance or more outside of the edge. The integration table with distance indexes is shown in Figure 3.3. Using a radially symmetric filter makes the pixel value orientation independent,
Figure 3.1: Lines consisting of four edges

Radially Symmetric Filter over two intersecting edges.

Figure 3.2: Constructing the Table

because it is the distance descriptors which are important. For thin lines, the value can be found by subtraction of the appropriate filter fragments (See Figure 3.4).
Figure 3.3: The Table

Figure 3.4: Thin Lines
3.2 Distance Metrics

The problem, then, is to keep track of the distances from the four edges of the line. Since this ordinarily involves square roots, another solution must be found. The solution is to find an appropriate distance metric which will give an approximate geometric distance. A curve is described by a function \( f(x,y) \) with \((x,y)\) on the curve when \( f(x,y) = 0 \). A commonly used alternative to geometric distance at point \( q = (x,y) \) is the value of \( f(q) \). Since \( f(q) = f(cq) \) for \( c \neq 0 \), \( q \) is first normalized in order to make the value meaningful. According to Vaughan Pratt, if we normalize \( f \) to \( \frac{f}{|\nabla f|} \) where \( \nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}) \), and \( f \) is linear, then the algebraic distance coincides with the geometric distance[Pratt87].

If a line is specified from \( x_1, y_1 \) to \( x_2, y_2 \) with width \( w \), then the equation of the line is \( f(x,y) = -(x_2 - x_1)y + (y_2 - y_1)x - x_1y_2 + x_2y_1 \). For a distance metric the equations are:

\[
\begin{align*}
\hat{f}_x &= \frac{\partial f}{\partial x} = y_2 - y_1 \\
\hat{f}_v &= \frac{\partial f}{\partial y} = -x_2 + x_1 \\
\frac{\hat{f}_x}{|\nabla f|} &= \frac{y_2 - y_1}{\sqrt{(f_x)^2 + (f_v)^2}} \\
\frac{\hat{f}_v}{|\nabla f|} &= \frac{-x_2 + x_1}{\sqrt{(f_x)^2 + (f_v)^2}}
\end{align*}
\]

The \( \frac{\hat{f}_x}{|\nabla f|} \) and \( \frac{\hat{f}_v}{|\nabla f|} \) can be considered the \( i \) and \( j \) components of the normal to the line. These components keep an accurate geometric distance from a line. If there is a unit pixel step away from the line in the \( x \)-direction,
then the perpendicular distance from the line is the value of the normalized \( f_x \). Each additional unit pixel step in \( x \) will add another normalized \( f_x \). Similarly each unit pixel step in \( y \) results in an addition to the perpendicular distance of a normalized \( f_y \). To keep track of the edges of the line consists of recording four distance boxes into which are added normalized \( f_x \) values for movement in \( x \) and normalized \( f_y \) values for movement in \( y \).

### 3.3 Anti-Aliasing a Straight Line

The algorithm is to compute one set of \((\frac{f_x}{|V|}, \frac{f_y}{|V|})\) components because the two side edges have the same normals while the end edges of the line have normal components of \((\frac{f_x}{|V|}, -\frac{f_x}{|V|})\) since they are perpendicular to the side edges. Starting at one corner of the line, the distances to two of the edges are zero, a width distance to the other side edge and a length distance to the other end edge (See Figure 3.5). For thin lines, an efficient method is to proceed in scan line order up the line using table lookups and compares to determine whether the pixel value is zero or not. A pixel value greater than zero means that the pixel is within the filter "footprint" of the line. The method's inner loop requires only the adds concerned with keeping the distance to the four lines, table lookups to determine the filter value based on those distances, and compares to determine whether you have reached one of the edge of the filtered image.

For thick lines an efficient scanning algorithm is to track the sides of the line and fill in the middle of the line with the maximum shade of the line. Also, between corners only one lookup is required into a one-dimensional
table because the edges will only affect each other when the scanned pixel is within a filter radius of two edges. The one dimensional table would be the values of a filter in respect to an infinite edge. Therefore a thick line is scanned with mostly one-dimensional table lookups and requires an efficient fill algorithm for fast operation.

3.4 Anti-Aliasing a Quadrilateral

The algorithm is easily adapted to convex quadrilaterals by computing the components for each edge making up the quadrilateral. Each edge would have its components for the normal to the edge. An efficient method for quadrilaterals is to trace the filtered edges and fill the center of the quadrilateral with the maximum shade. There is a loss in precision from the fact that the tables are computed from 90 degree intersections and a general quadrilateral can have its component lines meeting at arbitrary angles. A solution is to compute two more table, one for acute intersections
and one for obtuse intersections. A table for edges meeting at 45 degrees and a table for edges meeting at 135 degrees should be sufficient for visual purposes. In general, the loss of precision is visually negligible and can usually be ignored.

The quadrilateral has to be convex because the edges are defined as infinite edges. If a quadrilateral were concave, then some of the edges would be crossing through another edge in between endpoints of the quadrilateral. The distance descriptor for the crossing edge would flip from positive to negative while the algorithm was tracking another edge. This would feed erroneous information to the lookup tables with the effect of rendering an inaccurate object.
Chapter 4

Higher Order Curves

4.1 Equations and Expansions

To anti-alias conics and cubics requires more work and gives less accuracy. However, the results are very good within certain limitations. For the general conic and cubic, the equations are, respectively:

\[ f(x, y) = ax^2 + by^2 + cxy + dx + ey + g \quad (4.1) \]

\[ f(x, y) = ax^3 + by^3 + cxy^2 + dxy^2 + ex^2 + gy^2 + hx + iy + jxy + k \quad (4.2) \]

The Taylor series expansion of a function in two variables is:

\[ f(x, y) = f + \frac{\partial f}{\partial x}(x) + \frac{\partial f}{\partial y}(y) + \frac{1}{2} \frac{\partial^2 f}{\partial x^2}(x^2) + \frac{\partial^2 f}{\partial x \partial y}(xy) + \frac{1}{2} \frac{\partial^2 f}{\partial y^2}(y^2) + \ldots \]
For conics and cubics, the expansion has a finite number of terms. Then, since each step in \( x \) or \( y \) is a unit step, the values that each of the derivative is multiplied by is 1, 0, or \(-1\). For each step, the value of each derivative and the function \( f(x, y) \) can be kept incrementally with the use of additions.

To keep an incremental track of the value of a conic or cubic function and its derivatives, the functions must be determined as well as their values at the starting point.

### 4.1.1 Conic Function

For a conic, the initial value is taken by evaluating equation 4.1 at the starting point \((x_*, y_*)\). The Taylor Series expansion and derivatives of a conic are:

\[
\begin{align*}
  f(x, y) &= f + f_x(\Delta x) + f_y(\Delta y) + \frac{1}{2} f_{xx}(\Delta x)^2 + f_{xy}(\Delta x \Delta y) + \frac{1}{2} f_{yy}(\Delta y)^2 \\
  f_x &= 2ax + cy + d \\
  f_y &= 2by + dx + e \\
  f_{xx} &= 2a \\
  f_{yy} &= 2b \\
  f_{xy} &= c
\end{align*}
\]

Since \( f_{xx}, f_{yy}, \) and \( f_{xy} \) are constant, \( f_x \) and \( f_y \) can be kept incrementally by adding in the second derivative term with \( \Delta x = 1, 0, -1 \) depending on which way the step in \( x \) was taken and \( \Delta y = 1, 0, -1 \) depending on which
way the step in $y$ was taken. Then the first and second derivatives can be multiplied by their delta values and be added to the function value to get the function value at the next unit jump in $x$ and/or $y$ (See Figure 4.1).

The multiply is not needed because the $\Delta x$ and $\Delta y$ are either of unit magnitude or zero. Therefore keeping track of the function and its derivatives is given by taking the value of the function and its derivative at the starting point and incrementally adding the Taylor series components, and then updating the first derivative incrementally using the second derivative.

\[
\begin{align*}
  f_{\text{initial}} &= ax_x^2 + by_y^2 + cx_x y_y + dx_x + ey_y + g \\
  f_{x\text{initial}} &= 2ax_x + cy_y + d \\
  f_{y\text{initial}} &= 2by_y + dx_x + e \\
  f_{\text{new}} &= f_{\text{old}} + f_x(\Delta x) + f_y(\Delta y) + \frac{1}{2}f_{xx}(\Delta x)^2 + \\
                    &\quad + f_{xy}(\Delta x)(\Delta y) + \frac{1}{2}f_{yy}(\Delta y)^2
\end{align*}
\]
\[ f_{x_{new}} = f_{x_{old}} + f_{xx}(\Delta x) + f_{xy}(\Delta y) \]
\[ = f_{x_{old}} + 2a(\Delta x) + c(\Delta y) \]
\[ f_{y_{new}} = f_{y_{old}} + f_{yy}(\Delta y) + f_{xy}(\Delta x) \]
\[ = f_{y_{old}} + 2b(\Delta y) + c(\Delta x) \]

### 4.1.2 Cubic Function

From equation 2, a cubic function has 9 terms. This means that it has 9 derivatives with only three constant. It is still possible to keep track of a cubic's value and its derivatives' values, but it much more tedious.

\[ f(x, y) = f + f_x(\Delta x) + f_y(\Delta y) + \frac{1}{2} f_{xx}(\Delta x)^2 + \]
\[ f_{xy}(\Delta x \Delta y) + \frac{1}{2} f_{yy}(\Delta y)^2 + \frac{1}{6} f_{xxx}(\Delta x)^3 + \frac{1}{2} f_{xyy}(\Delta x)^2(\Delta y) + \frac{1}{6} f_{yyy}(\Delta y)^3 + \frac{1}{2} f_{xxy}(\Delta x)(\Delta y)^2 \]

\[ f_x = 3ax^2 + 2cxy + dy^2 + 2ex + hy + i \]
\[ f_y = 3by^2 + cx^2 + 2dxy + 2gy + hx + j \]
\[ f_{xx} = 6ax + 2cy + 2e \]
\[ f_{yy} = 6by + 2dx + 2g \]
\[ f_{xy} = 2cx + 2dy \]
\[ f_{xxx} = 6a \]
\[ f_{xyy} = 6b \]
\[ f_{xxy} = 2c \]
\[ f_{xyy} = 2d \]

21
Since \( f_{zzz}, f_{yyy}, f_{zyy} \) and \( f_{xuv} \) are constant, \( f_{zz}, f_{yy}, \) and \( f_{yz} \) can be kept incrementally by adding in the third derivative term with \( \Delta x = 1,0,-1 \) depending on which way the step in \( x \) was taken, and \( \Delta y = 1,0,-1 \) depending on which way the step in \( y \) was taken. Similarly, the incremental changes in the higher derivatives percolate up the line until every derivative has had its value updated incrementally. The function value can be kept by using the Taylor series sum.

The multiply is not needed because the \( \Delta x \) and \( \Delta y \) are either of unit magnitude or zero. Therefore keeping track of the function and its derivatives is given by taking the value of the function and its derivative at the starting point and incrementally adding the Taylor series components, and then updating the first derivative incrementally using the second derivative which in turn has been updated by the constant third derivatives.

\[
\begin{align*}
    f_{initial} &= ax_s^3 + by_s^3 + cx_s^2y_s + dx_sy_s^2 \\
                 &+ ex_s^2 + gy_s^2 + hx_sy_s + ix_s + jy_s + k \\
    f_{zinitial} &= 3ax_s^2 + 2cx_sy_s + dy_s^2 \\
                    &+ 2ex_s + hy_s + i \\
    f_{yinitial} &= 3by_s^2 + cx_s^2 + 2dx_sy_s \\
                    &+ 2gy_s + hx_s + j \\
    f_{zzinitial} &= 6ax_s + 2cy_s + 2e \\
    f_{yyinitial} &= 6by_s + 2dx_s + 2g \\
    f_{zyinitial} &= 2cx_s + 2dy_s \\
    f_{new} &= f_{old} + f_{zold} + f_{yold} +
\end{align*}
\]
\[
\frac{1}{2} f_{zz_{old}} + f_{zy} + \frac{1}{2} f_{vv_{old}} + \frac{1}{2} f_{vv_{old}} + \\
\frac{1}{6} f_{zzz}(\Delta x)^3 + \frac{1}{2} f_{zxy}(\Delta x)^2(\Delta y) + \frac{1}{6} f_{vvy}(\Delta y)^3 + \frac{1}{2} f_{vvy}(\Delta x)(\Delta y)^2
\]

\[
f_{v_{new}} = f_{z_{old}} + f_{z_{old}}(\Delta x) + f_{v_{old}}(\Delta y) + \\
\frac{1}{2} f_{zzz}(\Delta x)^2 + f_{zxy}(\Delta x)(\Delta y) + \frac{1}{2} f_{vvy}(\Delta y)^2
\]

\[
f_{v_{new}} = f_{v_{old}} + f_{v_{old}}(\Delta y) + f_{v_{old}}(\Delta x) + \\
\frac{1}{2} f_{vvy}(\Delta y)^2 + f_{zvy}(\Delta x)(\Delta y) + \frac{1}{2} f_{zvy}(\Delta x)^2
\]

\[
f_{z_{new}} = f_{z_{old}} + f_{zz}(\Delta x) + f_{zxy}(\Delta y)
\]

\[
f_{v_{new}} = f_{v_{old}} + f_{v_{old}}(\Delta y) + f_{zvy}(\Delta x)
\]

\[
f_{z_{new}} = f_{z_{old}} + f_{zxy}(\Delta x) + f_{zvy}(\Delta y)
\]

### 4.2 Getting the Normal

To use the anti-aliasing algorithm, we take the value of the function \( f \) and the partials \( f_z \) and \( f_y \) and normalize by dividing by \( \sqrt{(f_z)^2 + (f_y)^2} \). The normalized partials are now the \( i \) and \( j \) components of the normal to the curve at that point. The value of the function was normalized to give a fairly accurate distance away from the curve.

The algorithm then considers the curve as a straight line at that point and scans in the direction of the largest component of the normal. This is a **width scan**, since the purpose is to follow the center of a curve and then scan half a width on both sides of the curve. By scanning in the direction of the greatest normal, the linear approximation is more accurate since the distance to the edge is less than in the direction of the other.

23
normal. For example, if the partials had yielded components of $1i + 0j$, then the algorithm would scan in the $x$-direction until an edge had been reached on both sides of the central curve. Obviously, scanning in $y$ would yield erroneous values since the distance from the curve as recorded by the algorithm would not change, even though there would be many pixel steps and a large real distance from the curve.

When the partials are mostly equal, the scan goes diagonally. (See Figure 4.2). This requires that the scan be stepped out diagonally which requires that the anti-aliasing algorithm do another diagonal scan one pixel above or below the present one to cover all the pixels covered by the curve. If a curve following algorithm steps along the curve using both diagonal and vertical/horizontal jumps, then a diagonal scan may miss a diagonal band of pixels because of the manner in which the curve is being followed. When the $\frac{\partial}{\partial y}$ is dominant, the width of the curve is scanned in the $y$-direction and when the $\frac{\partial}{\partial x}$ is dominant the curve is scanned in the $x$-direction. Endpoints are considered in the same manner as line end edges.

The choices for the transition points for scanning direction were when the angle of the normal had passed the angles of: 35 degrees for switching from $x$ to diagonal, 55 degrees for switching from diagonal to $y$, 125 degrees for switching from $y$ to diagonal, and so forth. Switching is not done in terms of degrees, but by terms of the magnitudes of the partials. Also, each width scan overlaps the next orientation width scan slightly to eliminate the gaps caused by the switch of scan orientation.
Figure 4.2: Width Scans of a Curve

\[ f(x,y) = \text{central curve} \]

edge at a 1/2 width perpendicular to the central curve

overlap to eliminate gaps

filter footprint of curve
Chapter 5

Implementation

The algorithm was implemented using floating point numbers to calculate the derivatives and distances. These were then converted into 32 bit split integer format with 16 bits integer and 16 bits fractional. For the antialiasing values, there was a resolution of $\frac{1}{16}$ of a pixel based on the four most significant bits of the fractional part and a sign bit. If there were any significant bits in the integer part, the distance was larger than a filter radius away and the table was read accordingly.

The filter table was constructed by integrating a pillbox shaped filter over the intersection of two lines with the above pixel resolution of $\frac{1}{16}$ creating a 1024 entry table. The filter was of radius 1.0 pixels. For different filter radii, the 32 bit split integers were divided by the filter size. Each filter shape needs only one lookup table, regardless of the filter's absolute size. The filter size can be changed by scaling the indices to the lookup table. The most recent implementation of the line algorithm takes 40 milliseconds to render a 2 pixel wide, 200 pixel long, filter radius 1.0 line on the TI34010
Figure 5.1: Pillbox Radial Filter, radius 1 pixel

graphics processor.

The square root for the conic and cubic algorithm used two iterations of Newton's method after having divided by the position of the most significant bit. Since this is performed only once per width scan, the computational cost is bearable. The curve algorithm is approximately 10 times slower then the line method.
Chapter 6

Limitations and Future Work

6.1 What Does Not Work

One limitation is that there is no provision for filtering intersections of different lines or curves. At present, a simple maximum of the values is used. For a real filtered image of two intersecting objects, both would have an effect on the final image. This algorithm treats each object separately and use the maximum to produce a visually acceptable result. The intersection of quadrilateral edges have a defect that was covered in section 3.4.

The conic and cubic algorithm degrade for widths of 15 pixels or more because of the breakdown of the linear approximation. Also a high rate of curvature creates problems with both the linear approximation and the break between width scan orientations. The properties of width and radius of curvature are interdependent in determining when the algorithm breaks down. A very wide width breaks down a curve with a large radius of curva-
ture, while a very small radius of curvature will break down the algorithm for a thin curve. The defects show up in several ways. The most glaring is when the width scans do not overlap, since this produces a very visible gap in the curve. Another defect is that the linear approximation gives more and more erroneous distance measures the greater the real distance of a pixel from the curve. This defect shows up as a growth or decrease in the width of the curve when the curve has actual normals not 
13ce to the four sets of normals used by the width scans (in x, y, diagonal left, and diagonal right). If the radius of curvature is very small near a transition point for width scans, defects are exhibited in the filtered image by large breaks in the shading values.

6.2 Using the Radius of Curvature

A method that may correct the deficiencies of the curve algorithm involve using the radius of curvature. The Taylor series gives the values for $f_x$, $f_y$, $f_{xx}$, and $f_{yy}$, for free use. The equation for radius of curvature $R$ at a point on a curve is

$$R = \frac{1 + \left( \frac{dy}{dx} \right)^2}{\pm \frac{d^2y}{dx^2}}$$

To avoid the $\frac{3}{2}$ power, the term $[1 + (\frac{dy}{dx})^2]$ can be written another way. Using $\tan \theta = \frac{y}{x}$, then the inside term can be written $[1 + \tan^2 \theta]$ where $\theta$ is arctan $\frac{dy}{dx}$. Using a trigonometric identity, $[1 + \tan^2 \theta] = \sec^2 \theta$ which plugs
back into the original radius expression to eliminate the $\frac{3}{2}$ power

$$R = \frac{\sec^3 \theta}{\pm \frac{d^2 \theta}{dy^2}}$$

Since $\frac{d^2 \theta}{dy^2} = \frac{f_{xx}}{f_{yy}}$ in only requires a table constructed for values of $\sec^3 \theta$ which would be read by value of $\frac{dy}{dx} = \frac{f_x}{f_y}$ to find the information needed to determine the radius $R$.

One method of using this information would be to get a closer hit to the true distance from a curve. First, the radius of curvature on the curve must be determined. Then the point on the outside edge that is a $\frac{1}{2}$ width away perpendicular to the curve is determined. A circle with the radius of curvature osculates the curve at that point (See Figure 6.1). The equation for the circle is $x^2 + y^2 - R^2 = f(x, y) = 0$, with a function value of $v = f(x, y) = x^2 + y^2 - R^2$ if a point is not on the circle. If the point at which $f$ is evaluated has value $v$, then the distance to the circle can be determined.

$$v = f(x, y) = x^2 + y^2 - R^2$$
$$f(x, y) = x^2 + y^2 - R^2 - v$$
$$0 = f(x, y) = x^2 + y^2 - R^2 - v$$
$$x^2 + y^2 = R^2 + v$$

The quantity $R^2 + v$ is the radius squared of a circle with radius $R$ plus a value which is the actual distance away from the original circle that the point was evaluated at. To get this distance $d$ a Taylor series expansion is
used

\[ d = \sqrt{R^2} + v - R \]

\[ \sqrt{R^2} + v = h(v) = h(0) + h'(0)v + \frac{1}{2}h''(0)v^2 + ... \]

\[ \sqrt{R^2} + v = R + \frac{1}{2}v - \frac{1}{8}\frac{v^2}{R^3} + \frac{1}{16}\frac{v^3}{R^6} + ... \]

\[ d = -R + \sqrt{R^2} + v = -R + R + \frac{1}{2}\frac{v}{R} - \frac{1}{8}\frac{v^2}{R^3} + ... \]

\[ d = \frac{1}{2}v - \frac{1}{8}\frac{v^2}{R^3} + ... \]

For circles above radius 14 curvature, the first term is adequate, while sharper curvatures need more terms of the series. Given the radius of curvature and the determination of distance from the circle, it possible to keep a distance based on the curvature. The overhead for this method is keeping several tables of the convolving of the filter function with circles of different radius. The problems facing this solution is that their are gaps left by going out from the curve perpendicularly. The question is whether the gaps between going out perpendicularly can be made less than a pixel or if a gap can be filled by interpolation from the previous scan. More work needs to be done to see whether this method is viable.

6.3 Width Scans

Another method is to extend the concept of width scans. Presently, the curve algorithm scans vertically, horizontally, and diagonally because it is very simple to implement and use. A width scan could be done with a
Figure 6.1: Alternative Methods

Bresenham algorithm, but the presence of Moire patterns would seem to preclude such a method. However, if a table of Bresenham lines could be stored such that if drawn radially, they would completely fill a circle of radius 15, for example, then they could be used to implement an efficient width scan. The principle is to have enough Bresenham lines to fill a circle drawn radially without any gaps caused by missing a pixel. A line every degree could be sufficient. The Bresenham line would be selected out of a table by the components of the normal to the curve. The radius of curvature expression could then be used to determine how many lines would be needed to completely to width scan the curve from the points of the curve.
width scans made from a table of Bresenham lines drawn at a sufficient of incremental degrees so that a circle filled by the lines would be completely filled.

circle of radius \( n \) requires many lines to fill completely, but would do good width scans.

Figure 6.2: Proposed Width Scans of a Curve
Chapter 7

Conclusion

The algorithm presented in this paper yields fast exact results for lines, and fairly accurate results for higher order curves. The computational cost is low and the process yields an enhanced screen resolution with the incremental removal of aliasing artifacts. Without a need to resort to costly direct convolution or Fourier transforms, the calculations for anti-aliased lines can be measured in milliseconds. If hardwired, the time cost would drop even further. Curves are slower, but not unreasonably so. The method has a strong potential for interactive anti-aliased graphics given its speed and its enhancement of the apparent resolution. The renderings could be used as part of a toolkit such as the PostScript toolkit, but with higher screen clarity. The graphics could be used interactively because there would not be a long wait while the line or curve was rendered. The popularity of aliased line and curve rendering lies in their quickness. With the added benefit of anti-aliasing, a slower but better looking rendering is viable. In conjunction with anti-aliased text, slide quality pictures can be taken directly off
the screen. As an example in the results section, a figure is presented of a scanned picture of Africa over which anti-aliased text and graphics have been super-imposed. The algorithm warrants future development for more possible applications.
Chapter 8

Results

The following pages show the results of the work. All work was done on screens with $640 \times 480$ resolution. First is shown an aliased edge and a comparison of the filtered results from direct convolution with a digital pillbox filter and results from the incremental method. A triangle blown up 32 times was convolved with a circular pillbox filter of radius 16 pixels. The result was then sampled which shows up as the very small triangle in the top of the frame. The comparison is between the 90 degree corner done by direct convolution and the 90 degree corner drawn by the algorithm. A triangle was used because it took $\frac{1}{2}$ the time of a square to convolve.

Examples of incrementally rendered lines, quadrilaterals, conics, and cubics, along with a blow up of the frame is shown. An example of a high curvature, large width curve is shown to show the present limitations of the method.
Aliased Edge, Convolved edge on right, Sampled edge at top.

Figure 8.1: Before Filtering

Figure 8.2: Anti-aliased corner after direct filtering
Figure 8.3: Anti-aliased corner after incremental filtering

Anti-Aliased Lines and Quadrilaterals.

Figure 8.4: Lines
Closeup 20 pixel wide line with 1.3 pixel radius filter.

Figure 8.5: Lines

Circle, width 2, equation \(x^2 + y^2 - 2500 = 0\), filter radius 0.7
Ellipse, width 10, equation \(x^2 + y^2 + 2xy - 16900 = 0\), filter radius 0.8
Circle, width 5, equation \(x^2 + y^2 - 6400 = 0\), filter radius 1.2

Figure 8.6: Conics
Closeup of small circle and ellipse.

Figure 8.7: Conics

Parabola, width 3, equation \( .1x^2 - y = 0 \), filter radius 0.8
Parabola, width 5, equation \( -.1x^2 - y = 0 \), filter radius 1.2
Hyperbola, width 5, equation \( x^2 - y^2 - 1000 = 0 \), filter radius 1.0

Figure 8.8: Conics
Hangershape, width 5, equation \(0.0195y^3 + x^2 + y^2 - 400 = 0\), filter radius 1.0
Regular cubic, width 3, equation \(y^3 - 1000x = 0\), filter radius 1.0
Cubic on right, width 3, equation \(x^3 + y^2 - 10000 = 0\), filter radius 1.0

Figure 8.9: Cubics

Figure 8.10: Closeup of Cubics
Ellipse with high curvature on the major axis
Parabola of large width and high radius of curvature at the origin
Small circle which breaks down at gaps between width scans

Figure 8.11: Breakdown

Anti-Aliased Text and Graphics Over Africa.

Figure 8.12: Applications
Chapter 9

Appendix: Code Listing

9.1 Setting Up The Table

#define F0 0.0
#define F1 1.0
#define pi 3.14159

/*
   =========================================================================
   This program computes a pixel weight table of weight
given line distance. The table is of the form char[32][32]
where the indices are for 32 distances
ranging from -16 to 16.
The relationship is that 15 corresponds to 1.
The values are in fractions*255.
   =========================================================================
*/

/* This function computes the weight for a pixel given
two orthogonal line distances. d1 must be positive */
float fa(R, x, y)
float R, x, y;
{
float v, a;
if ((y*y + x*x) > (R*R)) v = FO; else
v = ( pi*R*R/4.0 ) +
    -((R*R/2.0)*asin(x/R) + (x/2.0) * sqrt(R*R - x*x) ) +
    -((R*R/2.0)*asin(y/R) + (y/2.0) * sqrt(R*R - y*y) ) +
    ( x * y );
return(v);
}

float fb(R, x, y)
float R, x, y;
{
return( fa(R,FO,y) * 2 - fa(R,-x,y) );
}

float fc(R, x, y)
float R, x, y;
{
return( fb(R,x,FO) * 2 - fb(R,x,-y) );
}
float fd(R, x, y)
float R, x, y;
{
return( fb(R,y,x) );
}

float distweight(R,d1,d2)
float R,d1,d2;
{
float v;
if ((d1 >= FO) && (d2 >= FO)) v = fa(R,d2,d1);
if ((d1 >= FO) && (d2 <  FO)) v = fb(R,d2,d1);
if ((d1 <  FO) && (d2 <  FO)) v = fc(R,d2,d1);
if ((d1 <  FO) && (d2 >= FO)) v = fd(R,d2,d1);
v /= fa(R,FO,FO)*4;

return(v);  
} 

main()  
{  
int i, j;  
char val[32][32];  
float j1, i1, k, radius;  
char fn[60];  

printf("radius?\n"); scanf("%f", &radius);  
printf("computing...\n");  

for (i=15; i>=-16; i--) {  
printf("\n");  
for (j=-16; j<=15; j++) {  
i1 = (float)i /15;  
j1 = (float)j /15;  
k = distweight(radius, i1, j1);  
printf("%d ", (int)(k*10));  
val[15-i][15-j] = (int)(k*255);  
}  
}  

printf("\n");  
printf("filename?\n"); scanf("%s", fn);  
fsave(fn, val, (long)32*32);  
}
printf(" width, x1, y1, x2, y2, fsize ?");
scanf("%f %f %f %f", &width, &x, &y, &m, &n, &fsize, &radius);

doaintline(width, x, y, m, n, fsize);
}

/*
   This function takes a starting point and an ending
   point and renders a line of width w with a filter
   of radius fsize between the two points.
*/

int doaintline(width, x1, y1, x2, y2, fsize)
float width, x1, y1, x2, y2, fsize;
{
    register long ea, ed, wxvy, wyvx;
    int i, j;
    float dx, dy, mag, factor;
    int x, y, xs, ys, value, tx, tval, val, initval;
    long offsetb, offsetc;
    long ta, tb, tc, td;
    int a, b, c, d;

    /* get the normal to the line */

    dx = x2 - x1; dy = y2 - y1;
    mag = sqrt((dx * dx) + (dy * dy));
    dx = dx / mag; dy = dy / mag;

    /* if the line has no width or length, return */

    if ((mag == (float) 0) || (width == (float) 0))
        return;

    xs = (int) (x1 + (width / 2.0 * (dy)));
    ys = (int) (y1 - (width / 2.0 * (dx)));

46
/*
set the factor to adjust the distance descriptors and the normal values based on the radius of the filter.
*/
factor = 65536.0 / fsize;

/* get the normals into proper split integer format */
wxvy = (long) (dy * factor); wyvx = (long) (-dx * factor);

/* set distances to the four edges, assuming that the starting pixel is at the intersection of two edges. */
e = 0L;
offsetb = (long) (width * factor);
offsetc = (long) (mag * factor);
ed = 0L;

/* uploop */
/* This part of the code handles tracing across the line and then preceding up the line, scanning across after every jump up, until the top of the 'footprint' of the line has been reached. */
x = xs; y = ys;
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
intval = value;

/* scan right */

while (value)
{
ea += wxvy; ed += wyvx; x++;
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
}
ea -= wxvy; ed -= wyvx; x--;
value = lookup(ea, ea + offsetb, ed + offsetc, ed);

47
/* scan left */

while (value)
{
/* go up a pixel */
ea += wyvx; ed -= wxvy; y++;
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
if (value)
{
ta = ea; td = ed;
tval = value; tx = x;

/* scan left */

while (value)
{
ea -= wxvy; ed -= wyvx; x--; 
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
val = ReadPixel(x, y);
WritePixel(x, y, MAX(value, val));
}
ea = ta; ed = td;
value = tval; x = tx;

/* scan right */
while (value)
{
val = ReadPixel(x, y);
WritePixel(x, y, MAX(value, val));
ea += wxvy; ed += wyvx; x++;
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
}
ea -= wxvy; ed -= wyvx; x--; 
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
continue;
}
/* go down one unit in x and y to see if we missed anything */
ea -= wyvx; ed += wxvy; y--; 
ea -= wxvy; ed -= wyvx; x--; 
value = lookup(ea, ea + offsetb, ed + offsetc, ed);

/*/ if value is zero, then top of line footprint reached; go to down loop to start scanning down. */
if (value <= 0) 
break;
}
hold();
/*/ downloop */

/*/ restore initial values as we are starting at same point except now we are scanning the line and going down in y. */
x = xs; y = ys;
value = tval = initval;
ea = ta = OL;
ed = td = OL;
tx = x;

/*/ scan right */

while (value) {
val = ReadPixel(x, y);
WritePixel(x, y, MAX(value, val));
ea += wxvy; ed += wyvx; x++; 
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
}
ea = ta; ed = td;
x = tx; value = tval;

49
/* scan left */

while (value)
{
    val = ReadPixel(x, y);
    WritePixel(x, y, MAX(value, val));
    ea -= wxvy; ed -= wyvx; x--; 
    value = lookup(ea, ea + offsetb, ed + offsetc, ed);
}

ea += wxvy; ed += wyvx; x++; 
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
while (value)
{

    /* go down one pixel in y */

    ea -= wyvx; ed += wxvy; y--; 
    value = lookup(ea, ea + offsetb, ed + offsetc, ed);
    if (value)
    {
        ta = ea; td = ed;
        tx = x; tval = value;
    }

    /* scan right */

    while (value)
    {
        val = ReadPixel(x, y);
        WritePixel(x, y, MAX(value, val));
        ea += wxvy; ed += wyvx; x++; 
        value = lookup(ea, ea + offsetb, ed + offsetc, ed);
    }
    ea = ta; ed = td;
    x = tx; value = tval;

    /* scan left */

    while (value)
{ 
val = ReadPixel(x, y);
WritePixel(x, y, MAX(value, val));
ea -= wxvy; ed -= wyvx; x--;
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
}
ed += wyvx; ea += wxvy; x++; 
value = lookup(ea, ea + offsetb, ed + offsetc, ed);
continue;
}

/* go up one unit in x and y to see if we missed any pixels 
in footprint of line */
ea += wxvx; ed -= wxvy; y++; 
ea += wxvy; ed += wyvx; x++; 
value = lookup(ea, ea + offsetb, ed + offsetc, ed);

/* if value is 0, then the line has been completely 
trace out */

if (value <= 0)
break;
}
}

/* This function looks up a value in the table based on 
the distance descriptors. The distance value is based 
on the four most significant bits of the fractional 
part, plus sign bit. If the distance has a significant 
integer bit, then the value is clipped to 0 if the 
distance was negative, and it is clipped to 31 if the 
distance was positive. */

int lookup(ea, eb, ec, ed)
long ea, eb, ec, ed;
{

int a, b, c, d, value;

/* The distance descriptor (ea,eb,ec,ed) is shifted right 16 bits, tested for integer bits, and then set to either 0, 31, or a value in between based on the four most significant fractional bits and the sign. */

a = (((ea >> 16) > 0) ? 31 : (((ea >> 16) <= -1) ? 0 :
    (((ea >> 16) == 0) ? 16 + (((ea >> 12) & 15) :
    ((ea >> 12) & 15)))));

b = (((eb >> 16) > 0) ? 31 : (((eb >> 16) <= -1) ? 0 :
    (((eb >> 16) == 0) ? 16 + (((eb >> 12) & 15) :
    ((eb >> 12) & 15)))));

c = (((ec >> 16) > 0) ? 31 : (((ec >> 16) <= -1) ? 0 :
    (((ec >> 16) == 0) ? 16 + (((ec >> 12) & 15) :
    ((ec >> 12) & 15)))));

d = (((ed >> 16) > 0) ? 31 : (((ed >> 16) <= -1) ? 0 :
    (((ed >> 16) == 0) ? 16 + (((ed >> 12) & 15) :
    ((ed >> 12) & 15)))));

/* The pixel value is based on the addition of two filter fragments and the subtraction of two filter fragments. The result is the value of the filter for the pixel we are at in respect to the four edges of the line. */

value = (wts[a][d] - wts[b][d] - wts[a][c] + wts[b][c]);
return (value);
}
Bibliography


