Polygon Statistics and Polygon Scan-Conversion and Antialiasing

Jim X. Chen and Xusheng Wang
Computer Science Department
George Mason University

Abstract

We present a new method for line drawing, triangle filling, polygon filling, and antialiasing. Current statistics show that our method improves line drawing and antialiasing about 6 times and polygon filling functions significantly over all existing methods. Specifically, it can improve finding the span extrema of polygons about 2 times and drawing the antialiased polygon edges more than 2 times faster than existing methods. The exact overall efficiency of polygon filling and antialiasing is a complex combination of many other factors, which we are unable to provide exact improvement statistics without hardware implementation and test.

The idea of our method is completely different from current existing graphics primitive scan-conversion (i.e., line drawing, triangle filling, quad filling, and polygon filling in general) algorithms. It is a great extension to our earlier work on line drawing [3, 5]. We save pixel patterns and distances of pixels to lines and edges in a table called SlopeTable. Based on this table, we provide the algorithms to improve the primitive scan-conversions and antialiasing. Improving the efficiency of graphics primitives will significantly advance the whole graphics research and application area.

1. Introduction

Polygon filling and line drawing are the most important graphics primitive functions. Most graphics models are filled polygonal objects or wireframe objects. Graphics hardware benchmarks and competitions all use polygon filling rate as their efficiency and capability indicator. Fast display and animation all require efficient polygon filling and line drawing. Improving the efficiency of polygon filling and line drawing will significantly advance the whole graphics research and application area. In graphics, many 3D graphics models are filled triangle meshes. Therefore, triangle filling is an important special case of polygon filling. Filling polygon with antialiasing is an associated time-consuming function. Significantly improving the efficiency of antialiasing will improve the quality of rendering results without sacrificing animation speed. Based on our statistics, about 88% triangles have all their edges shorter than 17 pixels. Therefore,
speeding up small triangle or polygon filling makes more sense.

Our method generally speeds up line drawing, triangle filling, quad filling, polygon filling, and antialiasing over existing methods. It is a significant improvement over existing methods and a big step forward in advancing graphics primitives and hardware. In the rest of the paper, we first introduce the statistics on triangle edges to show that most triangles have very short edges. Then, we discuss our new method, algorithms, and hardware implementations.

2. Triangle edge length distribution statistics

Mesa is a 3D graphics library originally developed by Brian Paul with an OpenGL command syntax and API. Most applications written for OpenGL can be compiled and run using Mesa without changing the source code. We analyzed Mesa’s source code, found all the entries that fill triangles, and inserted a capturing program segment at each of the entries to record the starting points and end points of triangle edges. We used the new generated library to compile and link graphics applications. When we execute an application compiled with our generated graphics library, we obtain all edges of triangles drawn by the application.

We collected 600 application programs in OpenGL. Then, our modified graphics library captured all triangle fillings in these programs and found that 416 of the 600 programs draw totally 1,414,858 triangles. The following tables list the statistical results. First, these 1,414,858 triangles have totally 4,244,574 edges. The length distribution of the edges are listed in Table 1. Here the number of pixels includes the two end points of each edge. We can see that 91% edges are shorter than or equal to 17 pixels ($dx$ is 16). Furthermore, more than 95% edges are shorter than and equal to 33 pixels ($dx$ is 32).

<table>
<thead>
<tr>
<th># of pixels</th>
<th># of edges</th>
<th>percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 &lt; # \leq 17$</td>
<td>3879974</td>
<td>91.410%</td>
</tr>
<tr>
<td>$17 &lt; # \leq 33$</td>
<td>182028</td>
<td>4.288%</td>
</tr>
<tr>
<td>$33 &lt; # \leq 65$</td>
<td>93756</td>
<td>2.209%</td>
</tr>
<tr>
<td>$65 &lt; # \leq 129$</td>
<td>49709</td>
<td>1.171%</td>
</tr>
<tr>
<td>$129 &lt; # \leq 257$</td>
<td>30157</td>
<td>0.710%</td>
</tr>
<tr>
<td>$257 &lt; # \leq 385$</td>
<td>6465</td>
<td>0.152%</td>
</tr>
<tr>
<td>$385 &lt; # \leq 513$</td>
<td>2390</td>
<td>0.056%</td>
</tr>
<tr>
<td>$513 &lt; #$</td>
<td>95</td>
<td>0.002%</td>
</tr>
<tr>
<td>Total</td>
<td>4244574</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

Second, we classify each triangle by the length of its longest edge. For example, the first row
(0 < # ≤ 17) in Table 2 means that each triangle’s longest edge is shorter than or equal to 17 pixels. The statistics shows that about 88.7% triangles have all three edges shorter than 17 pixels.

Table 2: Statistics on the number of triangles by the longest edge of each triangle

<table>
<thead>
<tr>
<th># of pixels</th>
<th># of triangles</th>
<th>percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; # ≤ 17</td>
<td>1255201</td>
<td>88.716%</td>
</tr>
<tr>
<td>17 &lt; # ≤ 33</td>
<td>80829</td>
<td>5.713%</td>
</tr>
<tr>
<td>33 &lt; # ≤ 65</td>
<td>39540</td>
<td>2.795%</td>
</tr>
<tr>
<td>65 &lt; # ≤ 129</td>
<td>20747</td>
<td>1.466%</td>
</tr>
<tr>
<td>129 &lt; # ≤ 257</td>
<td>14183</td>
<td>1.002%</td>
</tr>
<tr>
<td>257 &lt; # ≤ 385</td>
<td>3221</td>
<td>0.228%</td>
</tr>
<tr>
<td>385 &lt; # ≤ 513</td>
<td>1074</td>
<td>0.076%</td>
</tr>
<tr>
<td>513 &lt; #</td>
<td>63</td>
<td>0.004%</td>
</tr>
<tr>
<td>Total</td>
<td>1414858</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

From the above statistics, we learned that most triangles in computer graphics applications are very small. If we can speed up small triangle or polygon filling, we can significantly improve the efficiency of graphics drawing. In graphics, many 3D graphics models are tessellated as filled triangle meshes. Therefore, most polygons are really rendered as triangles.

3. New method for line drawing, polygon filling, and antialiasing

3.1. Existing methods

The general convex polygon filling algorithm includes three steps: a) sorting the edges by the Y values of the end-points, b) computing span extrema that lie between left and right edges for each scan-line, and c) filling all the pixels between each pair of the span extrema. Here, the first and the third steps are simple and straightforward, but the second step depends on the algorithm for calculating the span extrema.

Existing methods all use different incremental algorithms to find the span extrema. Dan Field’s algorithm (called B5 in [7], as shown in Appendix A here) is currently the most efficient algorithm that can be directly modified to calculate the span extrema. The modified algorithm (named B6 following Dan Field’s B5) is listed in Appendix B. For example, Fig. 1 shows the pixels chosen by algorithm B6 for filling the triangle with end points (2, 2), (14, 4) and (4, 14). The span extrema on each scan line are in grey, and the interior pixels are in black. The white pixels are not filled because it could overlap with a neighboring polygon. Actually, algorithm B6 will choose the white pixels on the right side of the triangle, but only fill the grey and black pixels.
3.2. Our new SlopeTable method

In an \( N \times N \) raster plane, we can divide all lines into \( N \) groups according to their slopes. Then we select a line which has the largest number of segments in each group to represent all lines in that group. Such a representative line in a group is called the Group Representative Line (GRL). We save the horizontal pixel length \( Q \) and the vertical pixel length \( P \) of each GRL in a table called SlopeTable. We discussed a SlopeTable method to scan-convert a line \([3, 5]\). The method significantly increased the number of segments of lines, and through copying pixel patterns and intensities, we can speed up line scan-conversion and antialiasing. Now, we extend the SlopeTable method to improve polygon filling and antialiasing. The idea is explained as follows.

We use the SlopeTable to save the pixel patterns of GRLs and distances from the pixels to the corresponding GRLs, so that we can find any line’s pixel patterns (for line scan-conversion), polygon span extrema (for polygon filling), and pixel distance (for antialiasing) without any calculation. Fig. 2 shows the SlopeTable’s basic structure. We only discuss lines and edges that have slopes between 0 and 1 (\( dx \geq dy \)). All other cases can be handled by symmetry. Next, we use a specific line as an example to explain the storage of pixel patterns.
As in Fig. 3, we have a line from (0,0) to (15,6). \((Q,P)\) for the line’s GRL is (5,2). The complete \(X\) pattern (namely \(XLP\)) is the line’s pixel pattern of \(Y\) changes along unit \(X\) increases. The first segment of the complete \(X\) pattern (called \(XP\)), which has \(Q\) bits, is actually saved in the SlopeTable. The complete \(Y\) pattern (namely \(YRP\)) is the line’s \(X\) run-length pattern along unit \(Y\) increases. (Because \(dx \geq dy\), when \(y\) increases 1, the \(x\) increment is equal to or greater than 1, which is called a \(X\) run-length.) The run-length pattern includes a short run-length that is \(dx/dy\) and a long run-length \(dx/dy + 1\). If we use 0 and 1 to represent the short and long run-lengths respectively, the \(Y\) pattern becomes \(YLP\). Here, we call the short run-length \(dx0\) and the long run-length \(dx1\). For the line’s GRL, \(dx0\) is \(Q/P\), and \(dx1\) is \(dx0 + 1\). The first segment of the \(YLP\) pattern (called \(YP\)), which has \(P\) bits, is actually saved in the SlopeTable. When \(y=0\), the run-length is equal to \(dx0/2\). We call this initial run-length \(dxx\).

When filling a polygon, we follow the scan line in \(Y\) unit increases to calculate the span extrema in \(X\) direction. If \(dy \geq dx\), \(XP\) is used to find the scan line’s span extrema. Otherwise, if \(dy < dx\), \(YP\) is used to find the scan line’s span extrema. To avoid computing \(dx0\), which involves a division for each edge, we also save \(dx0\) in the SlopeTable.

---

**Fig. 2: The SlopeTable method and its memory storage**

\(Q_i\) is the number of pixels in the first segment
\(P_i\) is the length in \(y\) direction of the first segment
In a $1024 \times 1024$ raster plane, for example, there are 1024 GRLs. All $XPs$ and $YPs$ have totally 50,574 bits and 25,287 bits, respectively. If we save each pattern with 16 bits (two bytes) in the SlopeTable, then $XPs$ and $YPs$ need 7,292 and 4,248 bytes of memory, respectively. If we save each pattern with 32 bits (four bytes) in the SlopeTable, then $XPs$ and $YPs$ need 8,376 and 5,532 bytes of memory, respectively. When the length of $XP$ or $YP$ (number of bits) is shorter than the provided hardware, the rest of the bits will repeat the corresponding pattern. In other words, some $XPs$ and $YPs$ are repeated in the memory, which can improve the efficiency of scan-conversion under certain situations.

For antialiasing, we need to know the distance from each pixel (in the pixel pattern) to the corresponding line. The distance is used to calculate the intensity of the pixel. Current methods calculate the distance iteratively in the scan-conversion. Our method can eliminate this calculation and improve the efficiency of antialiasing by saving all distances in the SlopeTable. Each GRL has $Q$ distances, so we need to save 50,574 distances for 1,024 GRLs. The signed distance $D$ of a pixel to its corresponding line is between -0.5 and 0.5. In our experiment, 17 different levels are enough to show a good resolution. If we use one byte to save a distance, the distance can be digitized to 256 different levels, which is more than enough.

The above design works for lines and polygon edges that have one pixel width. Antialiased lines usually have multiple pixel width. As we know, if the signed distance (negative if the pixel is above the line and positive if it is below the line) from the current pixel to the line is $D$, then the distance from the pixel above the current pixel to the line is $D - \cos \theta$, and the distance from the pixel below current pixel is $D + \cos \theta$. For each GRL, if we save $\cos \theta$ ($\tan \theta = dy/dx$) in the SlopeTable also, then
we can calculate antialiasing for multiple pixel width line easily. For 1,024 GRLs, there are 1,024 different \( \cos \theta \), which is between 0 and 1. We can also use one byte to save each \( \cos \theta \) value.

Fig. 4 shows the complete structure of the SlopeTable. We assign 16 bytes to each entry in the Main Table of the SlopeTable: 2 bytes for \( Q \), 2 bytes for \( P \), 2 bytes for \( dx0 \), 1 byte for \( \cos \theta \), 2 bytes for \( X \) pattern table pointer, 2 bytes for \( Y \) pattern table pointer, 2 bytes for distance table pointer, and 3 bytes not used. We added 3 empty bytes so that the next entry address index can be quickly obtained by 4 bit left-shift. Therefore, we need 16,384 bytes to save the Main table, 8,376 to save the \( X \) patterns, 5,532 bytes to save the \( Y \) patterns, and 50,574 bytes to save the pixel distances. In summary, we need 80,866 bytes memory buffer to save all the information for line drawing, polygon filling, and antialiasing, which is trivial in today’s computer hardware requirements.

**Main table** (16K byte ROM)

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( P )</th>
<th>( dx0 )</th>
<th>( \cos \theta )</th>
<th>( XP_0 )</th>
<th>( YP_0 )</th>
<th>( Dis_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q_0 )</td>
<td>( P_0 )</td>
<td>( dx0_{00} )</td>
<td>( \cos_{00} )</td>
<td>( XP_{00} )</td>
<td>( YP_{00} )</td>
<td>( Dis_{00} )</td>
</tr>
<tr>
<td>( Q_i )</td>
<td>( P_i )</td>
<td>( dx0_{i} )</td>
<td>( \cos_{i} )</td>
<td>( XP_{i} )</td>
<td>( YP_{i} )</td>
<td>( Dis_{i} )</td>
</tr>
<tr>
<td>( Q_{1023} )</td>
<td>( P_{1023} )</td>
<td>( dx0_{1023} )</td>
<td>( \cos_{1023} )</td>
<td>( XP_{1023} )</td>
<td>( YP_{1023} )</td>
<td>( Dis_{1023} )</td>
</tr>
</tbody>
</table>

**X pattern table** (8376 byte ROM)

**Y pattern table** (5532 byte ROM)

**Distance table** (50574 byte ROM)

3.3. New algorithm for calculating the span extrema

Based on the SlopeTable, we provide a new algorithm B7 in Appendix C that generates the span extrema. Compared to algorithm B6, B7 has almost the same overhead. In the for loop (line 5) in B7, when \( dy \leq 32 \), line 5.4.1 and line 5.4.2 will not be executed. B7 doesn’t have the additions (line 5.2.2 or line 5.2.4 in B6), instead it has a one-bit right shift operation (line 5.3). In hardware implementation, a one-bit shift operation is much faster than the addition operation. When \( dy > 32 \), B7 has a few additional operations including some test, increment, decrement, and data movement.
that can be implemented in hardware efficiently. As we mentioned earlier, statistics shows that more than 95% edges are shorter than 33 pixels. Note that we only need $dy \leq 32$ and there is no restriction on $dx$. Therefore, our algorithm should improve the calculation of the span extrema for more than 95% edges. Our statistics show that we can improve calculating the span extrema for a scan-line about 2 times.

3.4. Polygon filling with antialiasing

Algorithm B7 fills a polygon without antialiasing. With a little modification, we can integrate it with antialiasing. First, as shown in Fig. 5, we only fill the black pixels with algorithm B7. Then, we can use the SlopeTable again to find the pixel patterns and distances to scan-convert the edges with antialiasing. In our earlier work [5], we presented an algorithm (Alg2_multisegment) to draw an antialiased line that actually calculate the line’s pixel pattern and distances. Here, we can access the $X$ pattern and the distances in the SlopeTable directly to determine the pixel pattern and pixel intensities, which save much more time. Of course, the multiple segment copying method employed in the old algorithm (Alg2_multisegment) can still be exploited for speeding up drawing antialiased polygon edges. Statistics show that we can speed up about 6 times in line drawing with antialiasing, and 2 times in finding filling polygon edges with antialiasing.

The algorithm AntiEdge listed in Appendix D shows how to finish such a task. Compared to other line antialiasing methods (such as Gupta and Sproull’s method [8][9]), our algorithm is the only one that directly find the pixel pattern and distances, which significantly speeds up polygon edge scan-conversion and antialiasing. Polygon edge antialiasing is a little different from line antialiasing in deciding pixel intensities. As an example, we develop following two simple linear blending equations to calculate pixel intensities for polygon edges and lines, respectively. Given a pixel, let’s assume that its distance in the SlopeTable has 17 different levels between 0 and 16, the intensity of the corresponding pixel address in the framebuffer is $fcolor$, and the current pixel filling intensity is $ccolor$. If the pixel belongs to a polygon edge, we have the following blending equation to calculate the final antialiased intensity:

$$fcolor = (((16-distance)\times ccolor + distance \times fcolor)/16. \quad (1)$$

If the pixel belongs to a line, we have the following blending equation instead:

$$fcolor = ((8-|distance-8|)\times ccolor + |distance-8| \times fcolor)/8. \quad (2)$$
4. Hardware implementation

Fig. 6 shows the logical diagram for filling a polygon with the SlopeTable method. First, it calculates the span extrema for each scan line. Then, it generates all the X addresses for the pixels to be filled. The left side of the diagram is to find the left span extrema, and the right side is to find the right span extrema. These two parts are symmetrical. In the design, when we want to calculate the next span extrema, we only need one addition (Dan Field’s method [7] needs two additions.) Through the two MUXs, the saved pattern values can directly choose $dx_0$ or $dx_1$ for the addition operation. When an edge’s $dy$ is less than and equal to 32, we don’t need to fetch the pattern value from the SlopeTable again. Therefore, we prove again that this design is about two times faster than the existing methods when $dy \leq 32$.

In Fig. 6, MUX has three choices. MUX will initially choose $dx_{xx}$ to be added to the current X address. After that, MUX will choose $dx_0$ or $dx_1$ according to the lowest bit of the pattern in the 32 bit shift register. When the lowest bit is 0, $dx_0$ is chosen for the addition, otherwise $dx_1$ is chosen.
After one scan line is filled, X equals to Xr, therefore Comp1 will trigger the two shift registers to shift one-bit out to the 32 bit shift register. Then the next span extrema will be calculated. Comp1 will also trigger increasing the Y value. When Y gets to Yn, the current edge will be replaced by the next edge of the polygon. The process continues until all edges are scan-converted.

The two blocks for computing the next pattern addresses will only be triggered when dy is larger than 32. This part will generate the next pattern address when the end of the first segment is reached or the 32 bits have been shifted out of the shift register.

Fig. 7 shows the logical diagram for polygon edge or line antialiasing with the SlopeTable method. For each pixel of the first segment, the circuits generate its address (X, Y) and fetch the distance from the SlopeTable. The counters CT.X and CT.Y are used to generate the pixel address, which are initialized to X0 and Y0. Then CT.X will increase one by each pulse cp that is generated by cp0 and Comp. cp also shift the 32 bit shift register one-bit right to increase CT.Y 1 or 0 according to the bit value. At the same time, the pixel distance is fetched from the SlopeTable to the register.
Reg.Addr., which is used to calculate the intensities for the identical pixels in all different segments. This invention saves a lot of time in calculating the distance of a pixel from the exact edge.

![Diagram](image)

Fig. 7: The logical diagram for edge or line antialiasing

After getting the pixel address \((X, Y)\) of the first segment, it will be sent to the multiple segment copy engine to generate the addresses for the identical pixels in all segments through MUXx and MUXy. At the same time, for each pixel, the new intensity will be calculated in the compute intensity block.

When \(X > Xn\), the output of Comp will switch the choice of MUX, and generate a new \(cp\) to trigger the next round of calculation, which repeat the process above.

5. Error analysis and results

We discussed the errors of the old SlopeTable method in our earlier paper[5]. Our new statistics analysis show that the new SlopeTable method give the same pixel patterns as Bresenham’s and Field’s algorithms when a line or edges is shorter than 37 pixels. Since 95% of the polygon edges are shorter than 32 pixels, the new SlopeTable method has very small possibility to generate errors. The errors only happen with long lines. As we discussed in [5], long lime errors are neglectable.
For antialiasing, the error is undistinguishable and therefore can be ignored. Fig. 8 shows some images generated by the SlopeTable method and the existing method.

Fig. 8: Some images with different algorithms

**Reference**


Appendix A: Dan Field’s algorithm

Dan Field presented an incremental linear interpolation algorithm in his paper “Incremental Linear Interpolation” [7]. The algorithm can be applied to calculate the span extrema in most of the polygon filling algorithms. The following is the original algorithm B5 in Dan Field’s paper.

```
ALGORITHM B5
int i,a,b,n
int C1,C2,C3,C4,C5
int x,r
1 C1←b-a
2 C2←C1 div n
3 if C1≥0 then
     3.1 C3←(C1-C2*n)<<1
     3.2 r←C3-n
     3.3 C4←r-n
     3.4 C5←C2+1
else
     3.5 C3←(C2*n-C1)<<1
     3.6 r←C3-n-1
     3.7 C4←C3-n-n
     3.8 C5←C2-1
fi
4 x←a
5 for i=0 to n do
   5.1 output(i,x)
   5.2 if r≥0 then
        5.2.1 x←x+C5
        5.2.2 r←r+C4
   else
        5.2.3 x←x+C2
        5.2.4 r←r+C3
   fi
od
```
Appendix B: The modified algorithm for calculating the span extrema

Suppose a polygon has an edge which has two end points \((x_0, y_0)\) and \((x_n, y_n)\). According to the above incremental algorithm, with a little modification in line 4, we can have an algorithm below that calculates the span extrema. In this algorithm, \(a, b, C1\) and \(n\) are mapped to \(x_0, x_n, dx\) and \(dy\) respectively. We also rename \(C2\) and \(C5\) to \(dx_0\) and \(dx_1\). The algorithm is coded in C. Compared to algorithm B5, we add one more variable \(dxx\) which is assigned to \((dx_0/2)\) in line 4.1, and then added to \(x\) in line 4.2. This modification will let \(x\) have an exact initial span extreme value. When \(dy\geq dx\), this algorithm just gives the same pixel pattern as Bresenham’s line drawing algorithm. When \(dy<dx\), the algorithm generates the run-length of the line (or edge).

```
ALGORITHM B6(int x0, int y0, int xn, int yn)
{
    int dx, dy;
    int dx0, dx1, dxx, C3, C4;
    int x, y, r;
    dx=xn-x0;
    dy=yn-y0;
    dx0=dx/dy;
    if (dx≥0) {
        C3=(dx-dx0*dy)<<1;
        r=C3-dy;
        C4=r-dy;
        dx1=dx0+1;
    } else {
        C3=(dx0*dy-dx)<<1;
        r=C3-dy-1;
        C4=C3-dy-dy;
        dx1=dx0-1;
    }
    dxx=dx0/2;
    x=x0+dxx;
    for (y=y0; y<=yn; y++) {
        output(x, y);
        if (r≥0) {
            x=x+dx1;
            r=r+C4;
        } else {
            x=x+dx0;
            r=r+C3;
        }
    }
}
```
Appendix C: New SlopeTable algorithm for calculating the span extrema

Assuming a polygon edge’s two end points are \((x_0, y_0)\) and \((x_n, y_n)\), B7 is our new algorithm to calculate the span extrema based on the new SlopeTable. When \(dy \leq 32\), line 5.4.1 and 5.4.2 will not be executed.

```
ALGORITHM B7(int x0, int y0, int xn, int yn)
{
    int dx, dy, dx0, dx1, dxx, ny, y32;
    int x, y, i, j;
    unsigned int *patp, *patp1, r;

    dx=dxx=xn-x0;
    1.0 if (dx<0) dx=-dx;
    1.1 dy=yn-y0;
    3.0 if (dy≥dx) {
        3.1 i=(dx*2046+dy)/(dy<<1);
        3.2 patp=SlopeTable[i].XP;
        3.3 dx0=0;
        3.4 ny=SlopeTable[i].q;
    } else {
        3.5 i=(dy*2046+dx)/(dx<<1);
        3.6 patp=SlopeTable[i].YP;
        3.7 dx0=SlopeTable[i].dx0;
        3.8 ny=SlopeTable[i].p;
    } 3.9 dx1=dx0+1;
    4.0 if (dxx<0) {
        dx0=-dx0; dx1=-dx1;
    } 4.1 dxx=dx0/2;
    4.2 x=x0+dxx;
    4.3 patp=patp;
    4.4 r=*patp1++;
    4.5 y32=dy-32;
    4.6 if (y32>0) {
        i=ny; j=32;
    } 5 for (y=y0; y<=yn; y++) {
        5.1 output(x, y);
        5.2 if (r&0x01) 5.2.1 x=x+dx1;
        else 5.2.2 x=x+dx0;
        5.3 r=r>>1;
        5.4 if (y32>0) {
            5.4.1 if (--i==0) 5.4.1.1 { patp1=patp; i=ny; j=0; }
            else } 5.4.2 if (j==0) 5.4.2.1 { r=*patp1++; j=32; }
        }
    }
}
```
Appendix D: algorithm for polygon edge antialiasing

Assuming a polygon edge’s two end points are \((x_0, y_0)\) and \((x_n, y_n)\), the following algorithm \((\text{AntiEdge})\) achieves antialiasing using the SlopeTable. A simple linear blending function is used to calculate the pixel intensities. Different blending functions may be employed to achieve different antialiasing results.

```c
ALGORITHM AntiEdge(int x0, int y0, int xn, int yn)
{
    int dx, dy, q, p, dis, side, flag;
    int i, j, x, y, xm, ym;
    unsigned char cr, cg, cb, fr, fg, fb, C1;
    unsigned int *pat_p,*dis_p,r;

    side=0;
    if (yn<y0) { swapd(&x0,&xn); swapd(&y0,&yn); side^=1; }
    if (xn<x0) { x0=-x0; xn=-xn; flag=10; side^=1; }
    dy=yn-y0;
    dx=xn-x0;
    if (dx<dy) { swapd(&x0,&y0); swapd(&xn,&yn); swapd(&dy,&dx); flag++; side^=1; }

    i=(dy*2046+dx)/(dx<<1);
    q=SlopeTable[i].Q;
    p=SlopeTable[i].P;
    pat_p=SlopeTable[i].XP;
    dis_p=SlopeTable[i].Dis;

    x=x0; y=y0;
    r=*patp++;
    j=32;
    for (i=0; i<q; i++) {
        if (r&0x01) y++;
        x++;
        r=r>>1;
        if (--j==0) { r=*patp++; j=32; }
        dis=*dis_p++;
        C1=16-dis;
        xm=x; ym=y;
        do {
            if (side==0) {
                fr=(C1*cr+dis*fr)/16;
                fg=(C1*cg+dis*fg)/16;
                fb=(C1*cb+dis*fb)/16;
            } else {
                fr=(C1*fr+dis*cr)/16;
                fg=(C1*fg+dis*cg)/16;
                fb=(C1*fb+dis*cb)/16;
            }
            WritePixel(xm,ym,flag);
            xm+=q; ym+=p;
        } while (xm<=xn);
    }
}
```