EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

Scanline edge-flag algorithm for antialiasing

Kiia Kallio

Media Lab, University of Art and Design Helsinki UIAH

Abstract

In this paper, a novel algorithm for rendering antialiased 2D polygons is presented. Although such algorithms do exist, they are inefficient when comparing to non-antialiased alternatives. This has lead to a situation where the developers — and the end users of the applications — need to make a choice between high speed and high quality. The algorithm presented here however equals the performance of an industry standard non-antialiased polygon filling algorithm, while providing good antialiasing quality. Furthermore, the algorithm addresses the requirements of a modern 2D rendering API by supporting features such as various fill rules. Most of the research in antialiased 2D rendering has been proprietary work, and there is very little documentation about the algorithms in the literature. This paper brings an update to the situation by providing a thorough explanation of one such algorithm.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation;

1. Introduction

In the rendering of 2D vector shapes today — such as in SVG [FFJ03] — the filling algorithm should be general (i.e. support concave, self-intersecting polygons with holes), efficient, mathematically correct with subpixel accuracy, support even-odd and non-zero winding fill rules and high quality antialiasing, allow implementation also on mobile devices with limited processing power [Cap03] and be suitable for hardware implementation as well [Ric05].

The list of requirements is extensive, and the literature lacks descriptions of such algorithms. Even if the importance of antialiasing has been recognized years ago, lack of algorithms that are efficient and simple to implement has lead to the situation where antialiasing is considered as a costly extra feature. Even those applications that implement antialiasing need to sacrifice the quality for the sake of efficiency. In this situation, antialiasing is not considered possible in low-end graphics libraries, neither do developers — also on high-end platforms — consider it feasible to implement on their own.

2. Related work

The basis of the algorithm presented here is in the edge-flag algorithm presented by Ackland et al. [AW81] in 1981. The precise calculations required for subpixel accurate DDA implementation are described by Hersch [Her88]. In addition to the edge flag algorithm, there are other relevant polygon filling algorithms described in the literature. Most notable is the classic scan-line edge list algorithm [FvDFH90], a text book example of polygon filling.

None of these algorithms directly handle various fill rules required for rendering self-intersecting polygons. Neither are they suitable for antialiased rendering, except when using regular supersampling, i.e. the image is rendered to a bitmap with higher resolution and then scaled down with appropriate filtering.

Antialiasing has been researched a lot in 3D graphics. Typical solution for antialiasing in polygon-based 3D rendering is full screen antialiasing, where basically the whole frame buffer is rendered in higher resolution and then sampled down. In practice this is more complex, for instance multisampling is a method where the texturing and shading is done only once per pixel while z-buffer operations are done for each sample. With a suitable sampling pattern, a relatively low amount of samples can provide adequate level of antialiasing without sacrificing performance too much.

With the typical use cases of 2D vector graphics, for instance text rendering or user interfaces, the requirements for antialiasing are high. With 3D graphics, a lot of the detail in

the image comes from textures, and textures are antialiased with pre-filtering. With 2D vector graphics, majority of the detail comes from the geometry. The images are also often high contrast — for instance black text with white background. This means that methods that produce accptable results with 3D don't necessarily work for 2D vector graphics, as the required amount of samples is higher.

In 3D rendering the texture data is already filtered, so the higher sampling frequency is only needed at the polygon edges. Some antialiasing approaches use this observation for calculating antialiasing only for the discontinuities. Typically this is achieved by coverage masks, where the blending of color values generated from different polygons is required only if the coverage masks define partial fill area. The A-buffer algorithm by Carpenter [Car84] is fundamental work in this area. The algorithm doesn't handle transparency and can produce artifacts with z-ordering, so later the algorithm has been improved to support better sampling scheme and transparency by Schilling et al. [SS93], Winner et al. [WKP*97] and Jouppi et al. [JC99]. The algorithm presented in this paper is related to these techniques and can be also used for coverage mask generation.

It is also possible to approach the antialiasing problem by calculating the coverages analytically instead of sampling. When high antialiasing quality is required, sample-based approaches easily get expensive considering the bandwidth and memory usage. Also, even if the amount of samples is very high, sample based approaches have always some upper limit where issues like moiré patterns or ringing become visible. Sample based approaches merely shift these issues to higher frequencies, but never get rid of them completely.

Analytical approaches on the other hand try to calculate the exact pixel coverage of the polygon with mathematical analysis. The principal work in this area was done by Catmull [Cat78] in the late 1970's. Analytical approaches produce higher amount of tones at the polygon edges and don't suffer from sampling artifacts. However, analytical approaches are typically computationally expensive by requiring clipping at pixel level and between polygons. Analytical approaches also easily suffer from problems with numerical accuracy at the special cases, such as overlapping polygon edges, thus being more difficult to implement in practice.

More recently for instance Loop et al. [LB05] and Qin et al. [QMK06] have been using 3D shader hardware for analytical antialiasing. These experiments are however suitable only for a limited set of use cases and can't be extended to support the requirements of a general rendering API.

In general, sample based approaches are more robust and can handle things like self-intersecting shapes and exactly matching polygon edges well without complex processing. The algorithms also scale well to large amounts of polygon data, since the amount of required storage depends on the amount of pixels in the display buffer, not on the amount of input data. These properties also make the algorithms easier to implement in hardware.

3. The algorithm

This paper presents the algorithm in incremental order, starting from a basic implementation and explaining the implementation of each feature one at the time.

The algorithm works in image space. This can be a temporary canvas of the size of the filled area, or preferably a one pixel high buffer that has the width of the scanline. To better illustrate the basics of the algorithm, the approach using a temporary canvas is explained first. The algorithm uses sample-based approach for antialiasing.

3.1. Basics of the algorithm

The algorithm is based on the edge-flag algorithm by Ackland et al. [AW81]. The edges of the polygon are first plotted to a temporaray canvas by a complement operation. Then the polygon is filled from left to right with a pen whose color is toggled by reading the bits from the canvas. This is typically done with a 1-bit per pixel offscreen bitmap. Figure 1 illustrates the filling operation with edge-flag algorithm.

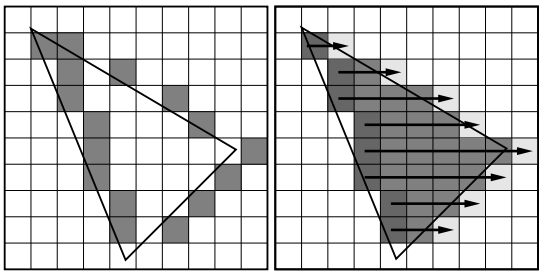

Figure 1: *A) Mark the edges B) Process the scanlines*

3.2. Re-organizing the bitmap

The edge-flag algorithm was originally developed for hardware implementation. Since the invention, there has been implementations on low-cost commercially succesfull products as well, such as Commodore Amiga [Com89]. In hardware, the access to individual bits can be done efficiently. However, the software implementations of the edge-flag algorithm have typically used the same screen buffer layout as the hardware. With 1-bit bitmaps where each byte contains an 8 pixel wide horizontal sequence, the processing either involves table-lookups or several shifting and masking operations, thus getting relatively slow, or the polygons need to be rasterized vertically, which causes sub-optimal cache performance.

However, when rasterizing antialiased polygons, one bit in a scanline doesn't map to one pixel on the screen, but

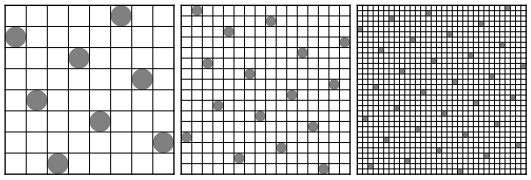

Figure 2: *Optimal n-rooks pattern with 8, 16 and 32 samples per pixel.*

to one sample within a pixel. By choosing suitably aligned amount of samples per pixel — for instance 8, 16 or 32 — it is possible to conveniently map the natural processsing units of the CPU to screen pixels. Now the bits in each unit are not seen as horizontal pixels next to each other, but as individual samples within a pixel.

The fill operation when performed this way is still a scan from left to right, but now instead of toggling individual pixels on or off, samples within a pixel are processed in parallel. For even-odd fill rule, this can be achieved with the exclusive or operation of the CPU.

3.3. Sampling pattern

Next step in extending the algorithm is choosing a suitable sampling pattern. The quality measures for sampling patterns have been researched mostly in the context of ray-tracing. With ray-tracing it's possible to choose unique sampling pattern for each pixel. Here a constant sampling pattern needs to used for the whole image.

A suitable pattern for the algorithm is n-rooks sampling pattern, i.e. all sample points in the pattern are distributed so that there is only one sample for each vertical and horizontal row. This is sometimes referred as sparse supersampling, for instance when used in InfiniteReality System by SGI [MBDM97]. As aliasing is mostly visible in nearly horizontal or nearly vertical edges, n-rooks pattern creates good results with such edges while keeping the amount of samples relatively low.

Since there are thousands of possible n-rooks configurations, the problem is in finding the optimal one. Discrepancy — as noted by Shirley [Shi91] — is a scalar measure of sample point equidistribution, and low discrepancy means better sampling pattern. Since n-rooks patterns are snapped to the sub-pixel grid, a relatively large set of sample configuration produces the same lowest discrepancy. For finding the optimal configuration from this set, stricter selection criteria is used: the sum of the squares of the minimum distances between the sample points. By driving this as low as possible, optimal n-rooks sampling patterns for 8, 16 and 32 samples are produced, illustrated in figure 2.

Also Laine et al. have developed a method for generating optimal sampling patterns [LA06]. This method however assumes free sample positioning and different weights per sample, and therefore can't be applied directly here.

Since there is one sample per each sub-scanline, and the subscanlines are evenly distributed, the pattern can be incorporated into the line plotting algorithm by modifying each sample position with a subpixel value. As the edge plotter is using a DDA, this adjustment is just a small offset value added to the horizontal position. Note that coordinates in the y direction need to be scaled by the amount of samples.

Algorithm 1 Plotting an edge with supersampling

1: $offset[8] \leftarrow 0.25, 0.875, 0.5, 0.125, 0.75, 0.375, 0, 0.625$
2: $x \leftarrow x_0$
3: **for** $y \leftarrow y_0, y < y_1, y \leftarrow y+1$ **do**
4: $\quad xi \leftarrow \text{FLOOR}(x + offset[y \bmod 8])$
5: $\quad bits[y][xi] \leftarrow \text{XOR}(1, bits[y][xi])$
6: $\quad x \leftarrow x + dx$
7: **end for**

Figure 3 illustrates the pixel access with the n-rooks sampling pattern when using subpixel scanline offsets.

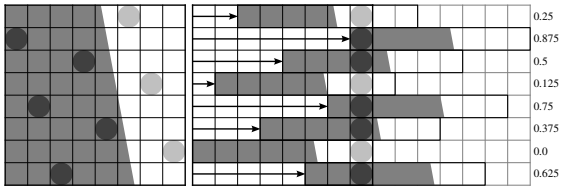

Figure 3: *A) Polygon edge crossing the supersampling pattern. B) Each scanline is offsetted by a fraction of a pixel to make the rounding of the DDA to produce correct plot positions.*

The fill operation doesn't know about sample positions; it just processes the data from left to right with XOR operation and generates a mask with one bit per sample for each pixel. This mask can be used for two purposes: it can be either converted to transparency by calculating the amount of bits [And05] or it can be directly used as a coverage mask with variants of A-buffer algorithm [Car84].

3.4. Non-zero winding rule

The algorithm presented this far already satisfies most of the requirements for an antialiased polygon fill algorithm. However, it only supports even-odd fill rule. If the application requires non-zero winding, plain edge-flag is not enough anymore, as the single on/off bit doesn't contain direction information of the edge.

In even-odd fill rule, the color of a pixel is determined by taking an infinite ray to arbitrary direction and calculating the amount of crossings it makes with polygon edges.

If the amount is odd, the pixel is filled, if it is even, the pixel is empty. With non-zero winding rule, the check includes a counter for the direction of the edges. For each clockwise edge, the value of the counter is increased and for each counter clockwise edge, the value of the counter is decreased. If the value of the counter is non-zero, the pixel is filled, if it is zero, the pixel is empty.

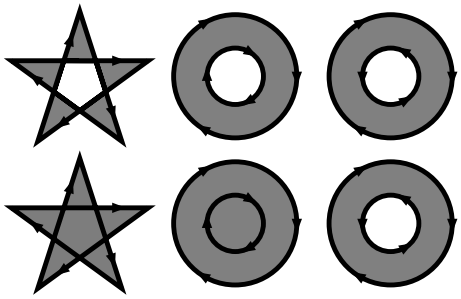

Figure 4: *Even-odd fillrule vs. non-zero winding fill rule.*

The algorithm can be extended rather easily to support non-zero winding fill rule. However, this is done at the cost of memory usage. Non-zero winding requires that instead of toggling a single bit on or off, a direction value of the edge is accumulated to a larger variable.

The principle is the same as when drawing with edge flags. The difference is that instead of just toggling a bit, the edge direction value (+1 or -1) is added to a temporary canvas during the edge plotting stage. The fill routine then accumulates the values from the canvas to a temporary variable, and whenever this changes to zero or away from zero, a bit in the coverage mask is toggled.

A related winding counter method has been used by Herf [Her97] for rendering soft shadow polygons. Also Aila et al. [AM04] have used similar method for rendering masks for occlusion culling.

When filling, processing the edge counters is much heavier than just toggling the bits with exclusive or. Now there is one temporary variable per sample instead of just one bit per sample. However, it is possible to combine this with scanning of the bit-per-sample buffer. This way it is possible to make the heaviest part of the processing conditional.

Since keeping a full integer received for the winding counter is not required in practice, it is possible to accommodate several values into a single variable. With 64 bits it's possible to represent 8 sample mask with 128 overlap levels. This can provide a significant speed impact since 64 bits can be processed with a single operation on some architectures.

In practice this is implemented by reserving an extra pad bit for overflow at every 8th bit, thus giving 7 bit counters, and keeping the pad bit empty by masking after the arithmetic operations that may cause an overflow.

3.5. Larger convolution base

The basic version of the algorithm provides only box filtered results. However, as the algorithm has a temporary canvas with higher resolution, it is trivial to extend it to use larger than one pixel weighted filter kernel for calculating the transparency of the final output pixel. For instance a 2×2 pixel base for the filter can be implemented relatively easily with table lookups.

A simple way to implement this is to process two scanlines in parallel, which means that each scanline will be processed twice. Another option is to save the filled mask values to a temporary buffer and re-use these when processing the next scanline.

The algorithm can also be easily modified to use different offset values for each color component, thus making the rendered image crisper on color matrix displays such as LCD panels [KdH03].

3.6. Scanline-oriented approach

The approach of using a large temporary canvas is not always possible. Especially if non-zero winding fill rule or higher than $8\times$ sampling is needed, the memory requirements of the buffer can be quite large. However, the algorithm does the filling operation per scanline anyway, only the edge plotting requires full height buffer. By adding an edge table to keep track of the edges it's possible to reduce the temporary canvas to have height of only one pixel. Therefore, as the final extension to the algorithm, the concept of large temporary canvas is replaced with a scanline-based approach. The implementation is straightforward and very similar to the classical scanline edge list fill algorithm [FvDFH90]. The major difference is that there is no need to keep the edges sorted. Also the amount of information stored per edge is slightly larger here.

The edge table (ET) consists of an array of slots, one slot per scanline of the target image. Furthermore, an active edge table (AET), is used for tracking the currently active edges. The edge table can be implemented most conveniently by using a linked list of edge table nodes. Each node needs to store the beginning and ending y scanlines in subpixel coordinates, current x value and x increment per subpixel scanline. If non-zero winding fill rule is supported, also a value for determining ascending/descending edge is needed.

When generating edges for a polygon, the starting y and the ending y are the first and last full subpixel scanlines that the edge crosses. The starting x is the x value at the crossing of the first scanline, and the x increment is $\frac{x_1-x_0}{y_1-y_0}$. The ascending/descending value depends on the direction of the edge. If y_1 is smaller than y_0, the direction value is set to -1, and the start and end points are swapped so that the edge initialization can be performed always assuming that y_0 is smaller than y_1.

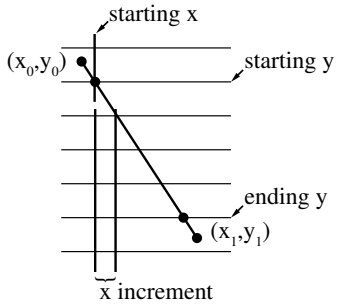

Figure 5: *Initializing the edge plotting procedure from the endpoints of the edge.*

Each edge is inserted to the edge table at a slot determined by starting *y*. The value of starting *y* is divided by the subpixel count (for instance 8) to get the correct slot. Note that edges that don't cross any scanlines are not inserted to the edge table.

When a scanline is processed, edges from the ET at the given scanline are moved to the AET. Then the edges in AET are plotted for to the sub-scanlines of the scanline. When the ending *y* of an edge is reached, the edge is discarded from AET. Last, the fill operation is done for the scanline, and the processing moves on to the next scanline.

3.7. Clipping

It is typical for polygon filling algorithms to require the input polygons to be clipped to the canvas prior to the rasterization. This is also the case with this algorithm.

The clipping can be done with a general polygon clipper if such is available. Since general polygon clipper that can handle self-intersection and holes is quite complex to implement, it is possible to bind the clipping operation to the rasterization.

First, the edges need to be clipped to the top edge and bottom edges of the clip rectangle. All edges that are completely above the top edge or below the bottom edge can be ignored. If an edge is crossing the top of the clip rectangle, a new *x* position should be calculated at the crossing and the starting *y* value set to the top of the clip rectangle. For the bottom edge, it is enough to set the ending *y* accordingly.

In horizontal direction, edges can't be directly clipped away. Instead, the edges that cross the left border should be clamped to the minimum value and the edges that cross the right border should be clamped to the maximum value. This can be implemented by splitting the edges that cross the borders to diagonal and vertical parts. If simplicity is favored over speed, it is possible to just add clamping comparisons to the edge plotting loop.

3.8. Optimizations

The algorithm is presented here emphasizing the clarity. The practical software implementation should consider some optimizations.

A. Fixed point arithmetic The DDA requires rational numbers, typically implemented using floating point arithmetic. Although floating point arithmetic is fast with modern processors, the floating point value needs to be converted to integer whenever an edge is plotted. This can be a slow operation, and much better results can be achieved by using fixed point arithmetic in the DDA. Even then, since the standard float to integer conversion may involve a performance hit on some processor architectures, better performance can be achieved if the floating point conversions involved in the initialization of the DDA are implemented with assembler.

B. Edge tracking The algorithm as presented this far performs the filling for the full width and height of the canvas. A very simple optimization to the algorithm is to add tracking for polygon extents. If minimum and maximum top and bottom values as well as minimum and maximum left and right values for the edges per scanline are stored, the fill algorithm can process only the area that is relevant for the polygon.

C. Mask tracking Typically the data at the temporary canvas is mostly zeros, as the data contains values only at the polygon edges. The filler loop can take benefit of this, and scan the data forward until a non-zero value is encountered. The scanning operation can be split to three cases: if current mask value is full coverage, the current color is applied directly to the target, if current mask is empty, pixels in the target are skipped, and if current mask has partial coverage, the color is blended to the target with appropriate alpha value. Whenever a non-zero value is encountered, only then is the mask updated. Also, as the non-zero value gets detected anyway, it is convenient to clear the value after it has been read. This way there is no need to perform the clearing separately.

D. Loop unrolling Since the edge plotting operates along scanlines, it is possible to unroll the plotting loops for full scanlines (a constant size sequence of subpixel scanlines) so that the plotted bit is defined with immediate data. Depending on the processor architecture, this can increase the performance for a few percents.

E. Scanline-oriented approach Surprisingly, switching from full buffer implementation to the scanline-oriented implementation can also give some speed increase. This is due to the cache behavior, as the scanline data can reside in the processor cache while it is being plotted to and used for filling. With the temporary canvas approach, especially the edge plotting stage is not optimal considering the cache usage.

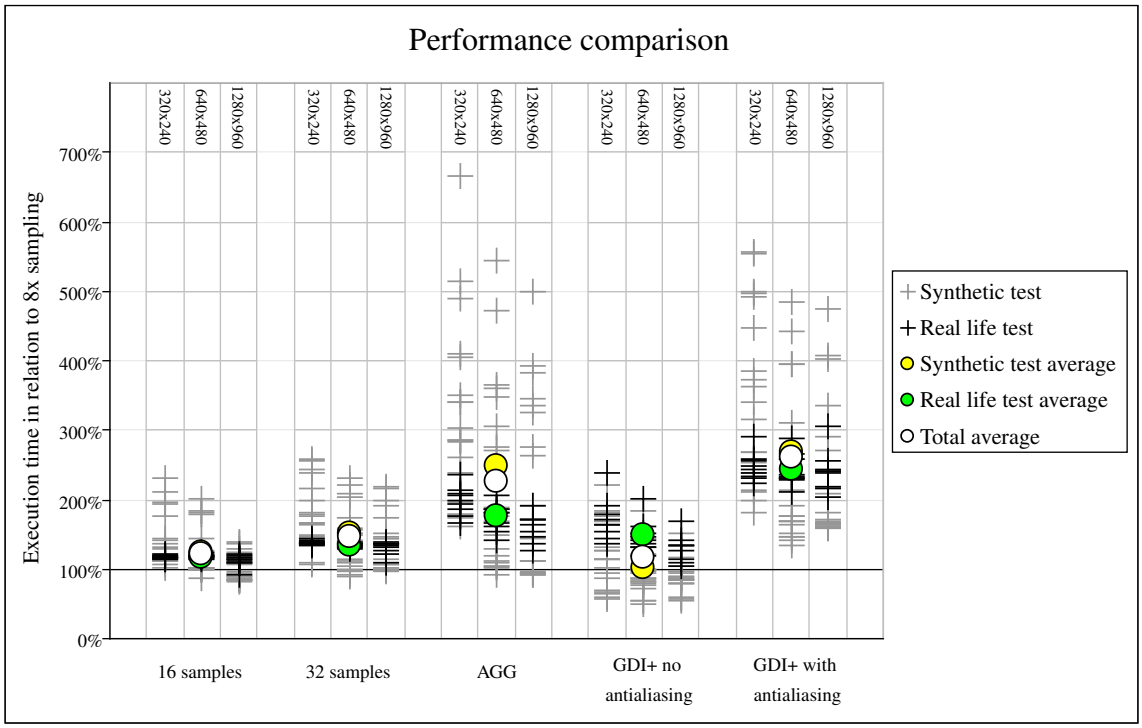

Figure 6: *Results of the performance comparison between the scanline edge-flag algorithm, AGG and GDI+.*

These optimizations are not processor specific but implementable with a high level language. In a performance-critical real life situation, the inner loops can be implemented directly with assembler language specific to the processor architecture. Since the algorithm is simple at that level, adding assembler optimizations is not an overwhelming task.

4. Results

The algorithm was implemented in C++ and compared against two 2D vector graphic libraries, GDI+ and Antigrain Geometry (AGG). GDI+ is a 2D rendering API developed by Microsoft for Windows platform. AGG is a cross-platform open source 2D rendering API focusing on performance and high quality. Three versions of the scanline edge-flag algorithm were evaluated in the comparisons; versions with 8, 16 and 32 sample points. GDI+ was evaluated both with and without antialiasing.

The executables for the comparison were compiled with Microsoft Visual Studio 7 Professional with identical optimization settings. The data sets for comparisons were also identical. All test data was polygonal, bézier curves were converted to polygons at load time to ensure identical amount of subdivision. The tests were run on a laptop with 2.4 GHz Pentium 4 M processor and 1.2 GB of memory.

The test data set included 29 images, 20 synthetic tests and 9 real-life images. The synthetic tests focused to benchmark the effects of one issue at a time, for instance number of edges, total length of edges or total filled area. The set of test images for synthetic tests contained 10 images, each was evaluated twice, once with even-odd fill rule and once with non-zero winding fill rule. The real-life images included vector illustrations with varying level of detail and tests for character rendering.

4.1. Performance comparison

Performance comparison was executed for three resolutions, 320×240, 640×480 and 1280×960. The test data was scaled to the screen resolution, and rendered for 200 frames while slowly modifying the translation and rotation. This procedure made sure that the underlying library couldn't use any caching of the rendered image. The timing for the performance included only the actual rendering calls, not the time spent for instance in display update.

Figure 6 contains the data from the performance comparisons. The data points are scaled relative to the test executed with $8\times$ sampling, thus a value on the 100% line indicates that the test in question executed in exactly the same time as the same test with $8\times$ sampling scanline edge-flag algorithm.

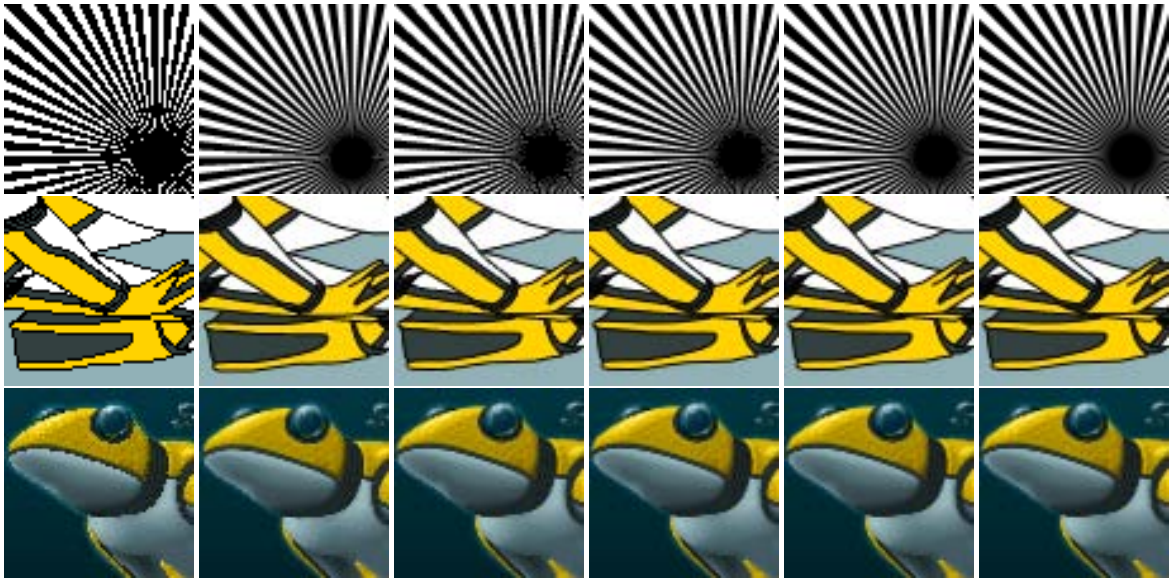

Figure 7: *Quality comparison of GDI+ non-antialiased, GDI+ antialiased,* 8× *sampling,* 16× *sampling,* 32× *sampling and AGG. Comparison images are available at* http://mlab.uiah.fi/~kkallio/antialiasing/

The performance tests indicate that on average the other antialiasing algorithms require 2× to 3× as much time as the scanline edge-flag algorithm. Even the version with 32 sample points per pixel is faster than the existing algorithms, having about 1.5× execution time when comparing to the 8 sample point version.

Only implementation that competes in the same performance range is non-antialiased rendering with GDI+. Even in this comparison the 8× sampling version of the scanline edge-flag algorithm is faster, especially when comparing the performance with real-life images. The performance of the non-antialiased GDI+ rendering is roughly equal to the 16× sampling version.

It has to be noted that the variation within the relative execution times is rather high. This is due to the different nature of the tests. The performance of the scanline edge-flag algorithm excels with complex shapes with lots of antialiased pixels. The difference in performance is not so great when testing for instance fill rate with a simple square polygon, as the performance inevitably becomes fill rate bound and the differences between algorithms don't have so big role.

4.2. Quality comparison

The image quality was evaluated from a subset of test images: one synthetic and two real-life images. The synthetic image displays a star of altering black and white pattern. The real-life images are vector graphic illustrations; one with low amount of detail and the other with high amount of detail.

The star pattern reveals the differences between analytic and sample-based approaches. The image rendered with AGG has no artifacts that result from point sampling. (Although not demonstrated in images here, it has to be noted that AGG can produce severe artefacts in some cases, for instance when edges are rendered on top of each other with even-odd fill and the same vertex coordinates. This is probably due to the numerical problems with the analytical approach.)

The antialiased rendering with GDI+ reveals that the sampling pattern is not a regular grid, but 8×4 grid with only 4 samples vertically. This is especially visible with near-horizontal edges: only four shades of grey are used for antialiasing.

Because of the n-rooks sampling pattern, the scanline edge-flag algorithm converts high frequencies to noise pattern. This is apparent especially for the 8× sampling version. When the number of samples increases, the results approach the image produced with AGG.

For the real-life images the difference is not that obvious. Basically all antialiasing improves the image quality dramatically. A close look reveals that GDI+ has weaker antialiasing quality for the vertical direction, for instance with near-horizontal lines and edges.

4.3. Conclusions

As a summary of the performance and quality comparisons, it can be stated that the algorithm presented here performs at least as fast as a standard non-antialiased algorithm, with quality similar to a standard antialiased algorithm.

If the rendering quality with 8 samples is not enough for demanding use cases, it is possible to reach a level of antialiasing similar to an analytic approach — but still with considerably better performance — by using a larger amount of samples.

5. Discussion and future work

The algorithm is designed for software implementation. A hardware implementation may need different design for best performance, and this requires further research.

Although this paper addresses the topic only within the field of 2D vector rendering, polygon filling algorithms are relevant in other research topics as well, for instance in soft shadow generation. Using this approach for polygon rasterization could produce better performance and quality of results also in those areas.

References

[AM04] AILA T., MIETTINEN V.: dPVS: an occlusion culling system for massive dynamic environments. *IEEE Computer Graphics and Applications 24*, 2 (2004), 86–97.

[And05] ANDERSON S. E.: Bit twiddling hacks, 1997-2005. Available at http://graphics.stanford.edu/~seander/bithacks.html.

[AW81] ACKLAND B. D., WESTE N.: The edge flag algorithm - a fill method for raster scan displays. *IEEE Trans. Computers 30*, 1 (1981), 41–48.

[Cap03] CAPIN T. (Ed.): *Mobile SVG Profiles: SVG Tiny and SVG Basic*. World Wide Web Consortium, January 2003. Available at http://www.w3.org/TR/SVGMobile/.

[Car84] CARPENTER L.: The A-buffer, an antialiased hidden surface method. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM Press, pp. 103–108.

[Cat78] CATMULL E.: A hidden-surface algorithm with anti-aliasing. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1978), ACM Press, pp. 6–11.

[Com89] COMMODORE-AMIGA, INC.: *Amiga Hardware Reference Manual*, 1.3 ed., 1989.

[FFJ03] FERRAIOLO J., FUJISAWA J., JACKSON D. (Eds.): *Scalable Vector Graphics (SVG) 1.1 Specification*. World Wide Web Consortium, January 2003. Available at http://www.w3.org/TR/SVG11/.

[FvDFH90] FOLEY J. D., VAN DAM A., FEINER S. K., HUGHES J. F.: *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Her88] HERSCH R.: Vertical scan-conversion for filling purposes. In *Proceedings CGInternational 88* (Geneva, 1988), Thalmann, (Ed.), Springer Verlag, pp. 318–327.

[Her97] HERF M.: *Efficient Generation of Soft Shadow Textures*. Tech. Rep. CMU-CS-97-138, CS Dept., Carnegie Mellon U., May 1997.

[JC99] JOUPPI N. P., CHANG C.-F.: Z3: an economical hardware technique for high-quality antialiasing and transparency. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 1999), ACM Press, pp. 85–93.

[KdH03] KLOMPENHOUWER M., DE HAAN G.: Subpixel image scaling for color-matrix displays. *Journal of the Society for Information Display 11*, 1 (2003), 99–108.

[LA06] LAINE S., AILA T.: A weighted error metric and optimization method for antialiasing patterns. *Computer Graphics Forum 25*, 1 (2006), 83–94.

[LB05] LOOP C., BLINN J.: Resolution independent curve rendering using programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM Press, pp. 1000–1009.

[MBDM97] MONTRYM J. S., BAUM D. R., DIGNAM D. L., MIGDAL C. J.: InfiniteReality: a real-time graphics system. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 293–302.

[QMK06] QIN Z., MCCOOL M. D., KAPLAN C. S.: Real-time texture-mapped vector glyphs. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), ACM Press, pp. 125–132.

[Ric05] RICE D. (Ed.): *OpenVG Specification Version 1.0*. The Khronos Group Inc., July 2005. Available at http://www.khronos.org/openvg/.

[Shi91] SHIRLEY P.: Discrepancy as a quality measure for sample distributions. In *Proceedings of Eurographics 91* (June 1991), pp. 183–193.

[SS93] SCHILLING A., STRASSER W.: EXACT: algorithm and hardware architecture for an improved A-buffer. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1993), ACM Press, pp. 85–91.

[WKP*97] WINNER S., KELLEY M., PEASE B., RIVARD B., YEN A.: Hardware accelerated rendering of antialiasing using a modified A-buffer algorithm. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 307–316.