Implementation of an OpenVG Rasterizer with Configurable Anti-aliasing and Multi-window Scissoring

Ren Huang and Soo-Ik Chae
School of Electrical Engineering
Seoul National University
{huangren,chae}@sdgroup.snu.ac.kr

Abstract

This paper describes an OpenVG-compliant hardware rasterizer with configurable anti-aliasing and multi-window scissoring. This rasterizer requires 129K logic gates with 2KB on-chip SRAM and provides satisfactory image quality with a reasonable rasterizer speed at the operational frequency of 100MHz. In this paper, we propose an optimized scanline algorithm, which provides better performance than the conventional scanline algorithm with supersample while maintaining the flexibility and the hardware simplicity. We also propose a fast LUT-based scissoring algorithm, which has zero-latency in most of the cases. The hardware implementation of this rasterizer is explained in detail.

1. Introduction

OpenVG [6] is a new royalty-free open standard API for hardware-accelerated two-dementional vector and raster graphics. Along with many other features, it provides the drawing functionality required by a SVG Tiny 1.2 viewer as well as some dynamic features for map display.

Among all stages of the OpenVG pipeline, the rasterization stage is very important, which usually contributes more than 50% of the rendering time. Rasterization in OpenVG is essentially filling polygons (probably complex and self-intersecting ones) [5]. The OpenVG specification defines three levels of rendering quality: NONANTIALIASED, FASTER, and BETTER, each of which uses a different anti-aliasing scheme.

The anti-aliasing techniques proposed in most of the literatures target at the 3D graphics applications. Some of them either require the polygons to be non-selfintersecting ([2],[10]), or employ difficult polygon decomposition to convert complex polygons into simple ones [4]. Even though others aim at or can be applied to 2D graphics, they are not optimized for hardware implementation [3] or they require large on-chip memories [8]. None of these methods is suitable for a low-cost vector graphics hardware rasterizer targeting at mobile devices.

This paper presents a OpenVG-compliant hardware rasterizer with the following features:

- It supports both odd-even and non-zero fill rules.
- It also supports two different anti-aliasing schemes as well as non-antialiased rendering to realize all the three rendering qualities.
- It uses an optimized scanline algorithm which provides better performance than the conventional one while maintaining the flexibility and hardware simplicity.
- It requires small on-chip memory (2KB).
- It has a small gatecount (129K) while it provides desirable image quality with satisfactory rasterizing speed at the operational frequency of 100MHz.

The rest of this paper is organized as follows. Section 2 explaines the optimized scanline algorithm, which is an extension of our previous work [5]), and describes its hardware implementation. Section 3 introduces the LUT-based scissoring algorithm used in our rasterizer, after explaining why we integrated scissoring into the rasterizer. Section 4 presents some images and their rasterizing time as references, which is followed by the conclusion.

2. Rasterization

2.1. Basic scanline algorithm with supersampling

In supersampling, more sample points are evaluated instead of using the pixel center as the only sample point, which is the case when anti-aliasing is disabled. Each sample point has some contribution (sample weight) to the intensity of this pixel (coverage value). At each pixel, every sample point is examined on whether it is inside the polygon or not. If it is, its sample weight is added to the coverage value of this pixel. For example, in Figure 1, at pixel *p0*, 6

out of 8 sample points (denoted as black dots) are inside the polygon, so the coverage of this pixel is 6/8 if each sample point has an equal weight of 1/8.

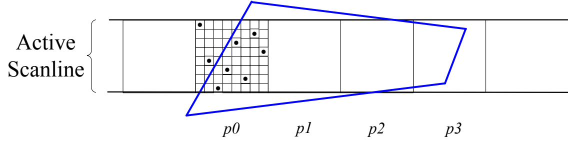


Figure 1. Illustration of supersampling.

To determine whether a sample point is inside the polygon or not, we draw a ray (conceptually) from this point to the left horizontally, and check all edges crossing this ray. Its winding count, which is initially zero, is increased by 1 if the direction of a crossing edge is upward, and decreased by 1 if downward. The odd-even rule says that a point is inside the polygon if its winding count is odd, while the non-zero rule reaches the same conclusion if it is not zero. Figure 2 illustrates the difference between these two fill rules when they are applied to a pentacle-shape polygon. OpenVG supports both odd-even and non-zero rules.

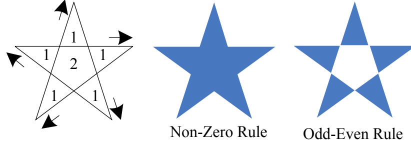


Figure 2. Illustration of two fill rules.

The sample pattern determines the number of sample points within one pixel and their positions. The N-Queens sample pattern used in this accelerator uses an $N \times N$ sample grid within a pixel and each sample point is placed such that no other sample point occupies the same row, column, or diagonal of the grid [7], as shown in Figure 1(8-Queens) and Figure 4(4-Queens). A weighting function $W(x_d, y_d)$ is used to determine the sample weight of each sample point, where x_d and y_d are the distances of this sample point from the pixel center along the x and y axes, respectively. Such weighting function is also called as a reconstruction filter in OpenVG specification. Two kinds of reconstruction filter are used in our rasterizer: box filter ($W(x_d, y_d) = 1$) and Gaussian $\frac{1}{2}$ filter ($W(x_d, y_d) = 2^{-4(x_d^2+y_d^2)}$). The box filter has an effective support radius (filter radius) of 0.5, which covers only one pixel; while the filter radius of Gaussian $\frac{1}{2}$ filter is 1.5, which covers 9 pixels, so the coverage calculation of a pixel should take the sample points in the neighboring pixels into consideration. Therefore, a Gaussian $\frac{1}{2}$ filter usually results in a lower rasterizing speed and a better image quality with smoother edges than a box filter does. The comparison of the resulting image quality and rasterizing speed of these two filters is given in Section 4.

2.2. Denotations

Before introducing our optimized scanline algorithm, some denotations and terminologies are introduced to facilitate its description.

- Active edge: an edge intersecting or totally lying inside the vertical sample range (e.g. AE1–AE4 in Figure 4).
- $minx, maxx$:as Figure 3 shows, the line on which an active edge lies has two intersections with the reconstruction filter; the $minx$ ($maxx$) denotes the x-coordinate values of the left (right) intersections.
- p_{cx}: the x-coordinate value of center of pixel p..
- pp_{cx}: the x-coordinate value of center of the pixel that precedes p.

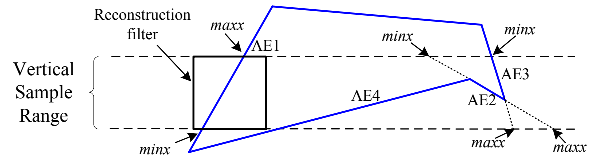


Figure 3. Illustration of the denotations.

2.3. Data structure

The data structure of an active edge in our algorithm includes the following items:

1) $AEy0$: y-coordinate of the lower vertex.
2) $AEy1$: y-coordinate of the upper vertex.
3) $AEx0$: x-coordinate of the lower vertex.
4) $AEminx$: ($minx$ - filter radius).
5) $AEmaxx$: ($maxx$ + filter radius).
6) $AEdy$: ($AEy1 - AEy0$).
7) $AEdx$: ($AEx1 - AEx0$), where $AEx1$ is the x-coordinate of the upper vertex.
8) $AEdirection$: indicates whether this edge is upward or downward.

While the necessary data structure to represent an active edge is $\{AEx0, AEy0, AEx1, AEy1\}$, the one used in our algorithm occupies 8 words, which is twice the size of the basic one. However, this structure enables us to avoid a lot of vain computations and memory accesses as discussed in Section 2.4 and Section 2.5, which speeds up the rasterization substantially.

2.4. Optimized algorithm

Though the basic scanline algorithm is easy to implement, it requires checking every sample point against every edge, which is too costly to be practical. Some observations help us optimize the algorithm, as introduced below.

Observation I. For all piexels on a certain scanline, only the active edges can affect their coverage values.

Optimization I. When a new scanline is to be processed, go through all the edges and put all the active edges into an active-edge table (AET). Sample points are checked against active edges instead of all edges.

Observation II. The inside-outside testing suggests that the active edges of interest are those that cross the conceptually horizontal ray drawn from the sample point to the left.

Optimization II. When the coverage value of a pixel p is computed, we ignore all the active edges with ($AEminx > p_{cx}$), for those edges are totally on the right side of current reconstruction filter. To skip the irrelevant active edges without going through all of them repeatedly, we should sort the active edges by $AEminx$ in advance.

Observation III. Only the active edges intersecting the filters applied to a pixel p and its neighboring pixels can make the coverage values of these two pixels different. For example, in Figure 4, active edge AE1–AE99 have the same effect on the winding counts of pixel $p1$ and $p2$; only AE100–AE103 make the winding counts of these two pixels different, which results in different coverage values.

Optimization III. When a pixel p is being processed, we record the winding counts when the first active edge with $AEmaxx$ greater than p_{cx} (denoted as AE_{start}) is encountered. [1] When the next pixel is to be processed, the examination starts from AE_{start} and the winding counts are accumulated based on the numbers stored previously. This process is illustrated in Figure 4 (assuming the winding counts before AE100 is checked are 0, 1, 0, 1 from top to bottom). As shown in Figure 4, instead of examining more than 100 active edges repeatedly, only a few active edges need to be checked at each pixel, which reduces the computation as well as the number of memory accesses substantially.

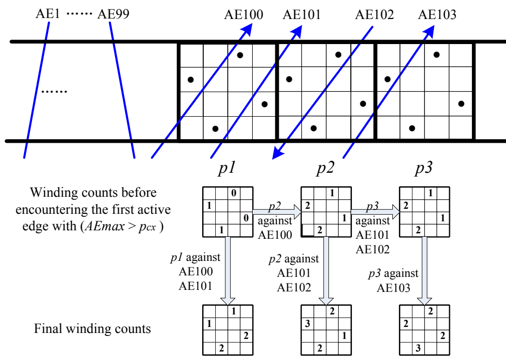


Figure 4. Illustration of ***Optimization III*****.**

Observation IV. Let the minimum $AEminx$ which is greater than p_{cx} be m; if pixel p is totally inside or totally outside the polygon, all pixels between p and $(\lceil m \rceil, p_y)$ have the same inclusion status with p.

[1] All active edges before this one are totally on the left side of the reconstruction filter without any intersection. The winding counts should be recorded before sample points are checked against this active edge.

Optimization IV. If a pixel p has a coverage value of 1 or 0, assign the same value to every pixel between p and $(\lceil m \rceil, p_y)$ without examination.

Putting all the optimizations together, the algorithm for filling pixels on a scanline is described as follows.

1) Go through all the edge data and construct an active edge table.
2) Sort the active edges by $AEminx$. Start to process the leftmost pixel from Step 3.
3) At the current pixel p, check each related sample point on whether it is inside the polygon or not. If it is, add its sample weight to the coverage value of this pixel. This step starts from the first active edge with ($AEmaxx > pp_{cx}$), and ends with the last active edge with ($AEminx < p_{cx}$). Winding counts are accumulated based on the numbers recorded when the previous pixel was processed. When the first active edge with ($AEmaxx > p_{cx}$) is encountered, record the index of this active edge and the winding counts for later use.
4) If the coverage value of this pixel is 0 or 1, assign the same value to every pixel between p and $(\lceil m \rceil, p_y)$, where m is the minimum $AEminx$ which is greater than p_{cx}.
5) Process the next pixel from Step 3 repeatedly until the horizontal bound of the current viewport is reached.

The algorithm presented above is a universal algorithm suitable for all reconstruction filters and sample patterns. To implement a new reconstruction filter or/and a new sample pattern, we only need to update the data of filter radius and sample positions as well as their sample weights. Since such information can be stored and altered easily in main memory, the anti-aliasing scheme can be configured by users as long as they provide valid parameters. This configurability enables flexible control of the trade-off between quality and rendering speed.

2.5. Hardware implementation

The rasterization stage was accelerated substantially on the algorithm level as described in the previous section. Here we describe the hardware implementation which accelerates it further by reducing the computation time as well as the memory accesses.

2.5.1. Reducing the computation time. To determine whether or not an active edge crosses the horizontal ray from a sample point (S_x, S_y) to the left, we need to find the x-intersection of this edge with the line on which the ray lies using the following equation:

$$x = AEx0 + \frac{(S_y - AEy0)(AEx1 - AEx0)}{(AEy1 - AEy0)} \quad (1)$$

The active edge intersects the ray if Sx is greater than x. This method requires five additions/substractions, one mul-

tiplication and one division. The following equation [9] is used to avoid the time-consuming division:

$$n = (S_x - AEx0)(AEy1 - AEy0) - (S_y - AEy0)(AEx1 - AEx0) \quad (2)$$

The active edge intersects the ray if $(n > 0)$. Since $(AEx1 - AEx0)$ and $(AEy1 - AEy0)$ can be pre-computed during the AET construction, they are included in the active edge data structure (denoted as dx and dy respectively). Hence the computation time is reduced further.

2.5.2. Reducing the number of memory accesses. The memory accesses mainly occur in the following three procedures: 1) going through all edge data to construct AET; 2) sorting active edges by $AEminx$; 3) reading active edge data when the sample points are checked against them. We discuss how the rasterizer accelerate these procedures in the following.

Reading edge data. Two buffers, each of which has sixteen 32-bit registers, are used to buffer the data. When the data in one buffer are being processed, the buffer controller fetches the next 16 words and stores them in the other buffer. To minimize the average memory access latency, 16-burst mode [1] is used. Our simulation shows that such double-buffering overlaps more than 94% of the memory access time with the computation time of AET construction, which results in a speed-up of 70%–80% in the Step 1 of our algorithm.

Sorting active edges by AEminx. Sorting requires extensive data movements with frequent memory access. We use a 2KB SRAM to buffer data and reduce the number of main memory accesses. In the rasterizer, sorting is divided into two stages. In the first stage, the data are read into the on-chip SRAM and sorted with a selection-sort algorithm and then written back to the main memory. After this step, all the data in the main memory are organized as sorted blocks, each of which has a size of 2KB. In the second stage, a merge-sort algorithm is used to merge all the sorted blocks in the main memory. By doing so, every relevant data item in main memory is accessed only $2(1+\lceil \log_2(\frac{N}{2048})\rceil)$ times, where N is the size of active edge table in bytes. The SRAM used this sorter is reused to cache the active edge data as introduced below.

Reading active edge data. Note that this procedure starts from the first active edge with $(AEmaxx > pp_{cx})$ and ends with the last one with $(AEminx < p_{cx})$. The data access pattern of this procedure is illustrated in Figure 5. Some active edges accessed when p_i was processed are accessed again when p_{i+1} is being processed, as shown in Figure 4 and marked by a shaded area in Figure 5. The SRAM used in the sorter is reused to cache such active edges. Note that the re-accessed active edges are always those that have been accessed most recently. Therefore, the SRAM is used as a 512-word cyclic cache which only stores the data of the latest 64 active edges. Two 32-bit registers ($initAddr$ and $endAddr$) are used to record the address of the oldest and the newest active edge data in the cache. A data item is in the cache if its address ($Addr$) is in the range of $[initAddr, endAddr]$, and its address in SRAM i can be calculated by the following equation:

$$i = ((Addr - initAddr)/4 + initSramAddr) \bmod 512 \quad (3)$$

where $initSramAddr$ is the SRAM address of the oldest data. If the data item is not in the SRAM, there are two possibilities: 1) the active edge requested precedes the latest 64 ones stored in the on-chip SRAM ($Addr < initAddr$); 2) this active edge has not yet been accessed. In the former situation, the data item is fetched from the main memory but not stored in the SRAM (we only cache the latest 64 active edges). In the later situation, the data fetched from the main memory should be stored in the SRAM, replacing the oldest data item (if the cache is full) or filling an empty entry (if otherwise); then $initSramAddr$, $initAddr$ and $endAddr$ are updated accordingly.

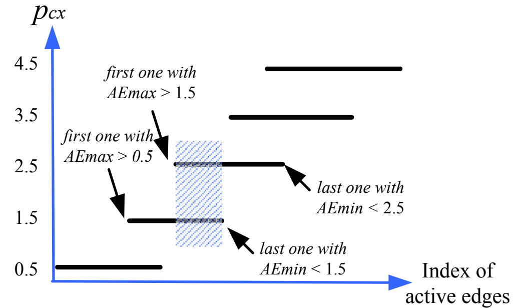


Figure 5. Illustration of data access pattern.

Two vector images (Figure 8(a) and Figure 6(c)) are used to evaluate the efficiency of the cyclic cache. Figure 8(a) represents an image with subtle details such as tiger's whiskers, while Figure 6(c) represents a text or a relatively sparse image. The profiling result is shown in Table 1, in which "#active edges" is the sum of the number of active edges on each scanline, and reading the data of one active edge (8 words) from main memory is counted as one memory access. As shown in Table 1, the number of memory access is reduced substantially and the hit rates excluding the compulsory cache miss are 100% and 91.3%, which is impossible to achieve by a conventional cache of the same size.

Table 1. Performance of the cyclic cache.

Figure	#active edges	#memory accesses		Reduction	Hit Rate
		no cache	with cache		
6(c)	8260	30110	8260	73%	100%
8(a)	64972	234769	70604	70%	91.3%

3. Scissoring

Drawing may be restricted to the union of a set of scissor rectangles. All OpenVG implementations are required to support at least 32 scissor rectangles. In Section 3.1, we explain the reason why scissoring is implemented in the rasterizer instead of in a stand-alone stage as the specification suggests. And in Section 3.2, an efficient look-up-table (LUT) based scissoring algorithm is introduced.

3.1. Scissoring in rasterizer

The OpenVG pipeline proposed by the specification suggests that the scissoring stage should be followed by the rasterization stage. While it is ideal in concept, it is not efficient in practice as the following discussion shows.

Scissoring and rasterization can be accelerated based by the following facts:

- We only need to check the pixels against the active scissoring rectangles, which are the scissor rectangles having intersection with the current scanline.
- If a scanline does not have intersection with any of the active rectangles, it is invisible, which means the coverage values of all pixels on this scanline do not need to be calculated.

If scissoring is implemented in a stand-alone stage after rasterization, the aforementioned optimizations cannot be performed so that the coverage value of every pixel (even it is on an invisible scanline) has to be calculated in the rasterization stage, and its position has to be checked against every scissor rectangle (even it is not an active one) in the scissoring stage, which results in a considerable waste of time. Therefore, instead of matching the proposed pipeline stage-for-stage, we integrate scissoring into the rasterizer to avoid vain computation. This integration eliminates the FIFO between rasterization and scissoring, which reduces the on-chip memory. However, it demands rapid scissoring scheme because rasterization and scissoring are no longer processed separately in parallel. The scissoring algorithm used in the rasterizer is introduced in the next subsection.

3.2. LUT-based scissoring

The most straightforward implementation of scissoring is checking a pixel against all active scissor rectangles, and if it is inside one of them, its coverage value is passed to the next stage; otherwise, it is discarded. In the worst case, when N scissor rectangles are used, a pixel has to go through N scissor tests, which takes at least N cycles excluding the memory access time. This computation load overweights the reduction of computation caused by the integration of two stages, so it is not suitable for our rasterizer. We use a LUT-based scissoring algorithm instead, which has zero-latency in most of the cases, as introduced below.

The basic idea of LUT-based scissoring is using a register as a look-up table (LUT), which records the scissoring status of a range of pixels, with each bit representing a pixel. If the corresponding bit of a pixel is set, the pixel is inside a scissor rectangle, so its coverage value is passed to the next stage; otherwise, it is discarded.

The LUT is constructed when a new scanline is to be processed. If a pixel that is currently processed is outside the range of the LUT, then the LUT is updated. A 64-bit register is used as the LUT, which records the scissoring status of pixel $p0$ – $p63$ initially and of pixels $p64$–$p127$, after an update. It will be reused for the next 64 pixels after each update. We use 64 parallel sub-circuits to examine the scissoring status of 64 pixels simultaneously so that it can update the LUT within one cycle. When there are N active scissor rectangles, it takes $(N + 1)$ cycles to construct or update a LUT. Note that the extra one cycle is used to clear the previous LUT before any active scissor rectangle is checked. The LUT construction/update can be done in parallel with the coverage calculation process, which reduces the performance overhead further and achieves zero-latency in most of the cases.

4. Experimental Result

Since no standard measuring metric for vector graphics rasterizer exists, we provide three types of images and their rasterizing time (Table 2) as references. Figure 6(a–d) represent vector text, Figure 7 is a typical animation-quality image, and Figure 8 is a high-quality static image.

The rendering time was obtained from HDL simulation on the following two conditions: 1) the rasterizer is simulated without the bus contention effect; 2) the initial main memory access latency and the access time of each word are assumed to be 4 cycles and 1 cycle, respectively.

Based on the image quality and rendering time of Figure 6(a–d), we chose box filter with 8-Queens sample pattern as the anti-aliasing scheme used for FASTER, and Gaussian $\frac{1}{2}$ filter with 4-Queens sample pattern for BETTER. The gatecount is 129K (synthesized with Synopsys© Design Analyzer, using 0.25μm standard cell library), excluding the 2KB on-chip SRAM.

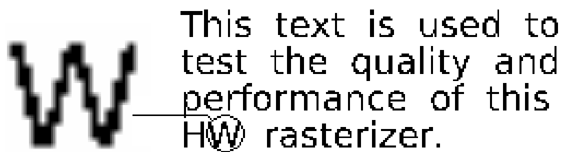
5. Conclusion

In this paper, we present a design of low-complexity hardware rasterizer targeting at vector graphics in mobile devices. It is fully OpenVG compliant and provides satisfactory image quality at a reasonable speed. An optimized scanline algorithm is used in this rasterizer, which provides

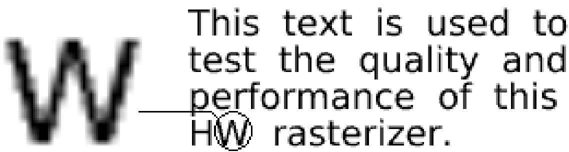
better performance than the conventional one while maintaining the simplicity and flexibility. Scissoring is integrated into the rasterizer to enable the optimization of both rasterization stage and scissoring stage. A fast LUT-based scissoring with zero-latency in most of the cases is introduced. This rasterizer can handle the data of more than 100 animation-quality images or 5 high-quality static images per second at a clock frequency of 100MHz.

Table 2. Rasterizing speed

Figure	#active edges	Rasterizing time (million cycles)
6(a)	2774	0.727
6(b)	8260	1.898
6(c)	8260	1.991
6(d)	13613	4.721
7	4092	0.982
8(a)	64972	19.638
8(b)	64972	18.392

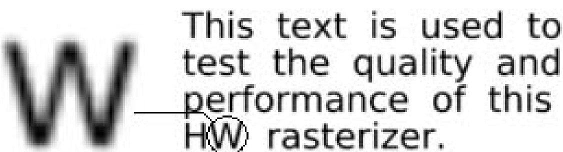


(a) Anti-aliasing disabled



(b) Box filter with 4-Queens

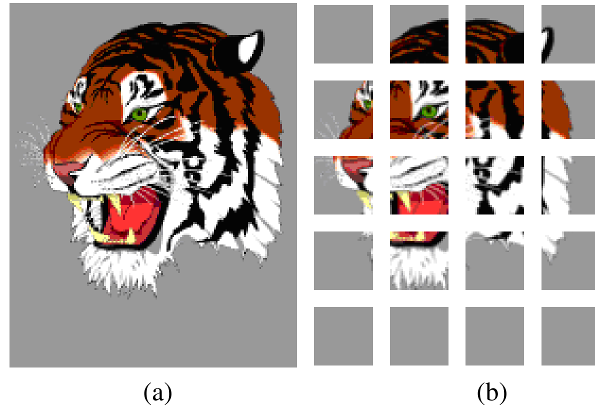
(c) Box filter with 8-Queens



(d) Gaussian 1/2 filter with 4-Queens

Figure 6. Text with different anti-aliasing schemes.

Figure 7. Courtesy of Lauri (FASTER).



(a) (b)

Figure 8. Tiger(FASTER)(a) with 20 scissors (b).

References

[1] ARM Ltd. *AMBA Specification (Rev 2.0)*. May 1999.

[2] A.Schilling. A new simple and efficient antialiasing with subpixel masks. *ACM SIGGRAPH Computer Graphics*, 25(4):133–141, Jul. 1991.

[3] K. Doan. Antialiased rendering of self-intersecting polygons using polygon decomposition. *Proc. 12th Pacific Conf. Computer Graphics and Applications*, pages 383–391, Oct. 2004.

[4] E.A. Feibush, M.Levoy and R.L. Cook. Synthetic texturing using digital filters. *ACM SIGGRAPH Computer Graphics*, 14(3):294–301, Jul. 1980.

[5] R. Huang and S.-I. Chae. Designing an OpenVG accelerator:algorithms and guidelines. *Proc. Int'l Conf. Computer & Communication Engineering*, pages 555–560, May 2006.

[6] Khronos Group Inc. *OpenVG Specification Version 1.0*. http://www.khronos.org/openvg/, Jul.2005.

[7] M.E.Goss and K.Wu. Study of supersampling methods for computer graphics hardware antialiasing. *HP Labs Technical Reports*, Dec. 2000.

[8] P.Haeberli and K.Akeley. The accumulation buffer: hardware support for high-quality rendering. *ACM SIGGRAPH Computer Graphics*, 24(4):309–318, Aug. 1990.

[9] P.J. Schneider and D.H. Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, Sep.2002.

[10] T.Diff. Polygon scan conversion by exact convolution. *Proc. Int'l Conf. Raster Imaging and Digital Topography*, pages 154–169, 1989.