An Optimized Rendering Algorithm for Hardware Implementation of OpenVG 2D Vector Graphics

Kilhyung Cha, Daewoong Kim, and Soo-Ik Chae
Electrical Engineering and Computer Science Department
Seoul National University
Seoul 151-742, Korea
{kilhyung.cha, dwkim316, chae} @sdgroup.snu.ac.kr

***Abstract*— An optimized rendering algorithm of the OpenVG 2D vector graphics for hardware implementation is presented in this paper. In the rendering algorithm we adopted a hybrid of raster and vector rendering, which uses vector rendering only within each scanline, to reduce both the number of external memory accesses and the computational complexity. We implemented a hardware accelerator with the proposed algorithm. Experimental results show that our hardware accelerator can handle 11.8 fps of Tiger image for a QVGA panel at the operating clock frequency of 100 MHz.**

Keywords-OpenVG, 2D Vector Graphics Hardware Accelerator

I. Introduction

Recently, the mobile applications using the 2D vector graphics such as SVG viewers, portable mapping applications, E-book readers, games, scalable user interfaces has been widely accepted in embedded devices because the 2D vector graphics has relatively smaller input data file size, provides lossless compression without artifacts and easy scalability for any target display size [1]. These features are provided at the cost of its increased computational complexity. For mobile devices, we need to reduce its power consumption by implementing it in hardware. The OpenVG^{TM} is a promising 2D vector graphics standard constituted recently by the Khronos group. It is a royalty-free, cross-platform API that provides low-level interfaces for vector graphics libraries such as Flash and SVG. Its primary applications are targeted for the handheld devices that require high-quality rendering with a limited power budget [2].

After an OpenVG-compliant reference software was firstly released in 2005 [3], several works on the optimized software algorithms have been reported [4][5]. However, their approaches were mainly focusing on reducing computational complexity as a sequential code without considering hardware implementation. Thus, they are not suitable for implementing an OpenVG hardware accelerator that satisfies the requirements of mobile devices. The purpose of this paper is to describe an optimized rendering algorithm for hardware implementation and its architecture that reduces the algorithm complexity and external memory accesses.

The rest of the paper is organized as follows. Section II briefly explains the OpenVG overview and describes the previous works. Section III proposes an optimized rendering algorithm suitable for the hardware accelerator. Experimental results obtained from RTL and FPGA simulation are summarized in section IV. Finally, the conclusion of this work is in section V.

II. Related Work

A. OpenVG Overview

Paths, paints, and images are the three types of basic components in the OpenVG 2D vector graphics. All the geometric objects to be drawn are defined by one or more paths, each of which consists of a sequence of segment commands and their corresponding coordinates. Each segment command in the standard format may specify a move, a straight line segment, a quadratic or cubic Bézier segment, or an elliptical arc. A paint command defines a color and a transparent effect, which is called a filtered alpha value (FAV), for each pixel being drawn, and images are rectangular collections of pixel effects such as texturing. Among the three types of the components, we need to better understand path drawing in order to find an efficient way of accelerating the rendering part. Users can fill or stroke a path, and each path segment described with a math formula is transformed into a series of edges through tessellation operations. Finally, the generated edges are displayed through rasterization process on a raster screen. Note that the number of edges in each path varies wildly, so the total number of edges in one image depends on its features. For example, Tiger, which is a representative test image of 2D vector graphics, has more than 220,000 edges and the number of edges in each path ranges from 2 to 40,000.

B. Previous researches

A hardware accelerator for the typical 2D vector graphics can be generally divided into two parts: a geometry part, which translates input commands and coordinates into geometrical objects, and a rendering part, which translates objects into pixel position and maps the proper colors [7]. It is especially more important to optimize the rendering part [8] because its computational complexity and memory bandwidth are substan-

This work was supported by “System IC2010” project of Korea Ministry of Knowledge Economy, Inter-university Semiconductor Research Center (ISRC) in Seoul National University, and the IC Design Education Center (IDEC) in KAIST.

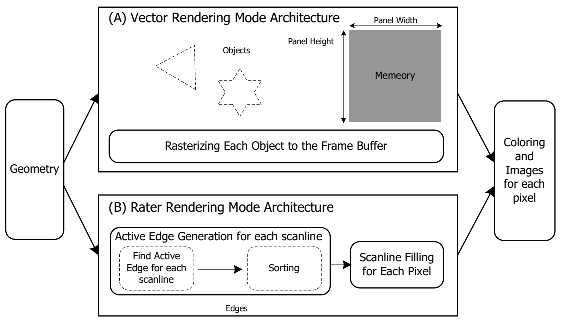
(A) Vector Rendering Mode Architecture
Panel Width
Panel Height
Objects
Memeory
Rasterizing Each Object to the Frame Buffer
Geometry
Coloring and Images for each pixel
(B) Rater Rendering Mode Architecture
Active Edge Generation for each scanline
Find Active Edge for each scanline
Sorting
Scanline Filling for Each Pixel
Edges

Figure 1. Two types of rendering mode: (a) a vector rendering mode, and (b) a raster rendering mode using scanline-based approach. A shaded rectangle in (a) represents the amount of memory requirement.

tially higher than those of the geometry part. The rendering part is often composed of two steps: object rasterizing and pixel processing.

We can classify 2D vector graphics rendering algorithms into two types: vector rendering, which draws each object every time in the frame buffer, and raster rendering, which draws each object by calculating the colors of its pixels in an image [9][10][11]. Architecture for the two rendering modes is illustrated in Fig. 1.

Raster rendering has an advantage compared to vector rendering because its algorithmic complexity is getting lower as the number of paths and the area enclosed by the path are increased. Thus, Most of the previous rendering algorithms for the 2D vector graphics are based on raster rendering [3][5][6]. It first searches all the edges in a path to find active edges and then sorts them in the scanning direction for each scanline. Then it calculates the coverage of a pixel based on the scanning direction in every scanline. However, this approach requires more computation because it should generally calculates all the parts which are not in fact displayed in a screen [13]. Furthermore, because sorting requires a lot of memory accesses, it could be the bottle neck in hardware implementation.

Vector rendering does not need to find and sort all the active edges but it computes the contribution of each object directly to the corresponding pixels according to the order of edges described in a path and accumulates them in a panel-sized buffer, which can be implemented as either off-chip or on-chip. A memory buffer allocated in an off-chip memory generally increases power consumption due to the large number of the external memory accesses while an on-chip memory buffer requires a large silicon area. It is prominently important especially if non-zero fill rule or super-sampling for anti-aliasing is supported. Therefore, the vector rendering architecture is not suitable for low-power applications

Although raster rendering does not require a larger memory buffer, it requires more external memory accesses for finding and sorting the active edges. Therefore, although its computational complexity is relatively low [5], the raster rendering approach is not suitable for hardware-based low-power applications [12].

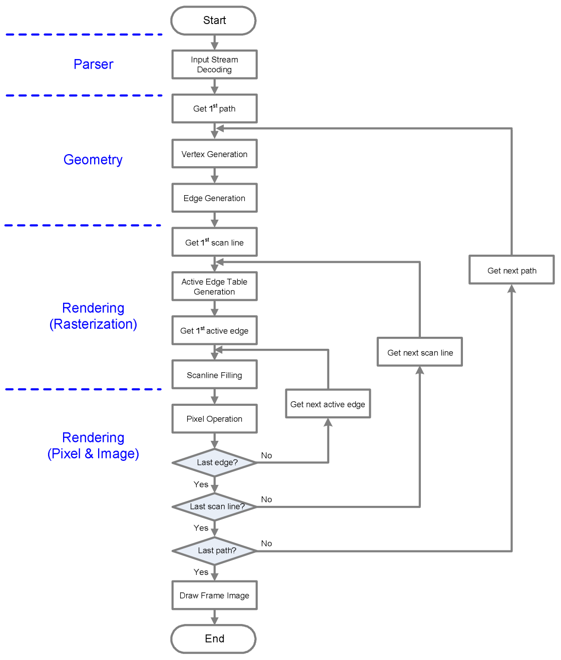

Figure 2. Flowchart for proposed 2D vector graphics rendering algorithm

We employed a hybrid of raster and vector rendering, which uses vector rendering only within each scanline. Consequently, it removes unnecessary memory accesses associated with sorting process and also reduces the size of required memory buffer. Therefore, it needs only a scanline-sized buffer, which can be implemented as an on-chip buffer. Moreover, it does not calculate the coverage of the sample points that are not affected by the edge. For this purpose we defined the cases of pixels intersecting with the edge into four categories, and this optimization scheme reduces most of the unnecessary edge calculations in scanline filling.

III. PROPOSED ALGORITHM

A. *Proposed Rendering Algorithm*

The proposed rendering algorithm starts with performing vector rendering for each scanline. Thus, it does not perform a scanline filling process by pixel-by-pixel but by edge-by-edge. Since edge-by-edge rasterizing does not require any sorting process, this approach can substantially reduce the bandwidth of its external memory accesses.

The proposed rendering algorithm consists of three steps: active edge table (AET) generation, edge function (EF) calculation, and FAV calculation. The AET generation gathers only the edges that cross a scanline, thereby substantially reducing the number of edges to be examined. This step is not covered in details in this paper since it is described in the previous works [3][6]. The second step evaluates the intensity of each sample point by performing EF calculation, which determines whether the edge is on the left side or not, and updates the partial winding values into the corresponding memory buffer for all the pixels affected by each selected edge. After completing the EF calculation for all the edges in the AET, we can finally calculating FAVs by summing up the intensities of sample points for each pixel. Fig. 2 shows the overall flowchart of the proposed rendering algorithm. Scanline filling corresponds to the EF and FAV calculation in our algorithm.

Now we compare the computational complexity of the proposed algorithm to the raster rendering algorithm. The AET generation is required in the proposed algorithm as like the raster rendering algorithm. Since the proposed algorithm does not need AET sorting process, however, the proposed algorithm reduces the complexity of at least $O(N \log_2 N)$, where N is the size of the AET. For the scanline filling stage, the proposed algorithm is more complex because it visits the same pixel repeatedly, whereas pixel-based raster rendering just visits one pixel once. However, the proposed rendering algorithm stores temporal winding counts, which are the result of EF calculation, in the local buffer, and then calculate the FAV just once for each pixel. Thus, the total number of EF and FAV calculation is the same. Moreover, the proposed algorithm eliminates the redundant EF calculations which are required in raster rendering [13], and apply further optimization that may not be applicable in raster rendering; more details for this optimization are stated later in this section.

For hardware implementation, the number of external memory accesses should be minimized to reduce power consumption. Thus, we compare the number of external memory accesses in evaluating the proposed algorithm. Table I describes the number of memory accesses for both algorithms per scanline. In the raster rendering algorithm [3][5][6], the number of read accesses to the active edges for scanline filling is increased in proportion to the number of sample points for anti-aliasing filtering, and the repeated read accesses to the active edge can be increased, which depends on the distribution of the edges. On the other hand, the proposed rendering algorithm requires the constant number of read accesses to the active edge regardless of the sampling quality and the edge distributional because it calculates the EF edge-by-edge. However, because the proposed algorithm calculates the EF edge-by-edge and store all the temporal winding counts for each pixel in the local buffer, thereby visiting one pixel repeatedly according to the range of each active edge and sample points affected by them. An external memory access for active edge generation step is same for both algorithm, and sorting is not required for proposed algorithm.

B. *Further Optimization*

The partial winding value of a pixel contributed from an active edge is obtained by evaluating the EF for all sample points in a pixel. This optimization scheme reduces the number of EF calculations by limiting them only for the active samples whose y-coordinates are within the range of the y-coordinates of the active edge.

We can classify the cases of pixels intersecting with an edge into four categories: (a) fully active, (b) partially active, (c) edge-embedded and (d) non-intersecting as shown in Fig. 3.

TABLE I. EXTERNAL MEMORY ACCESS ANALYSIS

	Raster Rendering [6]	**Proposed Rendering**
Active Edge Generation	M * Read + N * Write	M * Read + N * Write
Active Edge Sorting	N * (Read+Write) + N *$\log_2$N* (Read+Write)	None
Scanline Filling	N * Q * Read + $X_{repeated}$ * Read	N * Read

M: the number of edge for one path

N: the number of active edge

Q: the number of sample points for one pixel

$X_{repeated}$: the repeated access number depend on active edge's distribution

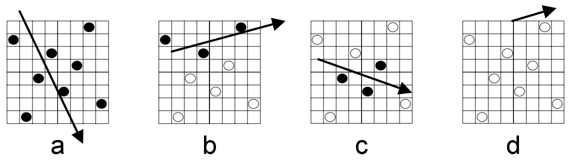

Figure 3. Active sample distribution in the examples for the four categories of edge pixels

Since we employed an equal-weight 8-Queen box filter for anti-aliasing filtering, only the active samples, which are represented with a filled circle, are evaluated for each edge. This optimization can not be applicable to the pixel-based raster rendering algorithm because it requires searching all the contribution edges per pixel at a time, thus we can not figure out which sample points can be skipped. According to our experimental results, it substantially improves performance since most edges intersects with only a small part of each pixel.

IV. EXPERIMENTAL RESULTS

We implemented a hardware accelerator for the 2D vector graphics with the proposed algorithm as shown in Fig. 4. For anti-aliasing filtering, we used an equal-weight 8-Queen box. We assumed a QVGA display panel and used a SDRAM memory controller for the external memory. For the bus architecture, we used an AMBA^{TM} AHB bus using a burst-4 transfer mode to minimize the memory access latency. For the number representation in the rendering part of the 2D graphics engine, we used a 24-bit fixed-point number system in which the integer and fractional parts of a number are represented by

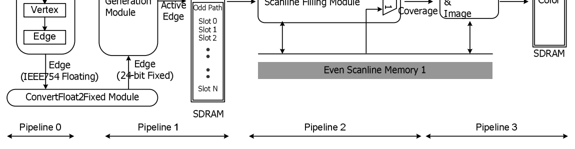

Figure 4. Architecture of Implemented 2D Vector Graphics Engine

10 and 13 bits, respectively and its sign by 1 bit. With the 24-bit fixed-point number system we reduced the complexity of the rendering part and improved its performance without losing the accuracy. Note that the edge information generated in the geometry part is represented as the IEEE754 single-precision 32-bit floating point standard, is immediately converted into the 24-bit fixed point as shown in Fig. 4.

To evaluate the performance of the proposed accelerator, we have used eight test images, whose features are summarized in Table II. Tiger and Dude are representative test images released from Khronos Group [2] and Hybrid Graphics Ltd. [3], respectively, and the others are translated from SVG files by the authors.

In Table III (a) we compare the number of external memory accesses for the raster and the proposed rendering algorithm. The experimental results show that the proposed algorithm reduces the SDRAM accesses by 52% to 82%; it implies that the proposed rendering algorithm is more efficient than the raster rendering algorithm for low-power systems. Table III (b) shows the proposed optimization scheme reduces the number of EF calculations by 36% to 83%, each of which takes 4 or 5 cycles. We found that most edges are crossing with only a small part of each pixel, which implies that the raster rendering algorithm performs a lot of unnecessary calculations. Table III (c) describes the estimated performance of the hardware accelerator. Assuming that the operating clock frequency is 100MHz, it can render 11.8 Tiger images per second for a QVGA display panel while a previous works [6] can handle 5 fps.

TABLE II. FEATURES OF TEST IMAGES

Test Image	Number of Path	Number of total Edges	Max. Edge per Path	Max. Active Edge per Scanline
Tiger	305	225,297	40,433	2,209
Dude	167	83,171	15,835	5,609
E-Book	19	44,955	3,105	817
Picture	712	50,820	464	136
Basket	719	77,904	232	136
Bottle	899	46,565	128	59
Snow	481	16,922	632	136
Pelican	95	4,027	722	136

TABLE III. MEMORY ACCESS COMPARISON AND PERFORMANCE

Test Image	(a) SDRAM access (x 10^6 in byte)		(b) the number of EF Calculation		(c) FPS @ VGA, 100MHz
	Raster Rendering	*Proposed Rendering*	*without Opt.*	*with Opt.*	
Tiger	413.61	197.95	2,179,640	496,320	11.8
Dude	115.98	39.31	487,960	82,528	40.5
E-Book	66.12	12.98	971,312	198,231	33.4
Picture	81.28	28.17	721,104	309,208	20.9
Basket	71.74	13.10	574,496	168,432	40.3
Bottle	51.71	10.45	537,312	204,597	36.0
Snow	26.58	8.60	267,688	124,189	45.2
Pelican	15.74	7.04	163,376	104,407	77.8

V. CONCLUSIONS

In this paper, we proposed a hybrid rendering algorithm for the 2D vector graphics, which uses vector rendering only within each scanline. Experimental results show that it substantially reduces both the external memory accesses and the computational complexity, which make it suitable for low-power high-performance graphics engine. We implemented a hardware accelerator with the proposed rendering algorithm, which can render more than 11 fps for Tiger QVGA image at the operating clock frequency of 100 MHz.

REFERENCES

[1] K. Pulli, "New APIs for mobile graphics," Proceedings of SPIE – The International Society for Optical Engineering, vol. 6074, 2006

[2] Khronos Group Inc., OpenVG 1.0.1 Specification. [Online] Available: http://www.khronos.org/openvg/.

[3] Hybrid Graphics Forum, OpenVG Reference Implementation (2005), http://forum.hybrid.fi

[4] G. He, B. Bai, Z. Pan, and X. Cheng, "Accelerated rendering of vector graphics on mobile devices," Lecture Notes in Computer Science, vol. 4551, pp. 298-305, 2007.

[5] S. Lee, S. Kim and B. Choi, "Vector graphics reference implementation for embedded system," Lecture Notes in Computer Science, vol. 4761, pp. 243-252, 2007.

[6] R. Huang and S. Chae, "Implementation of an OpenVG rasterizer with configurable anti-aliasing and multi-window scissoring," Proceedings of the Sixth IEEE International Conference on Computer and Information Technology, pp. 179-184, 2006.

[7] J. Foley, A. vanDam, S. Feiner, and J. Hughes, Computer Graphics: Principles and Practice, 2nd ed., Addison-Wesley, 1990.

[8] T. Mitra and T. Chiueh, "Three-dimensional computer graphics architecture," Current Science, vol. 78, pp. 838-846, April 2000.

[9] A. Schilling, "A new simple and efficient antialiasing with subpixel masks," ACM SIGGRAPH Computer Graphics 25(4), pp. 133-141, 1991

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

[11] K. Doan, "Antialiased rendering of self-intersecting polygons using polygon decomposition," Proceeding of 12th Pacific Conf., Computer Graphics and Applications, pp. 383-391, 2004

[12] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. D. Man, "Global communication and memory optimizing transformations for low power signal processing systems," in VLSI Signal Processing, vol. VII, pp. 178-187, 1994.

[13] S. Harrington, Computer Graphics A Programming Approach, 2nd ed., McGraw Hill, New York, 2006