A Design of a Mobile Graphics Accelerator based on OpenVG 1.0 API

Jae-Chang Kwak, Kwang-Yeob Lee, *Seokyeong University*

Abstract—In this paper, we propose the hardware architecture to accelerate 2D Vector graphics process for mobile devices. we propose the Transformation Unit Architecture that considerates the operation dependency. It has 3 cycles excution time and uses 2 multipliers and 2 adders. Proposed paint generation unit uses a LUT method, so it does not execute color interpolation which needs to be calculated every time. The proposed OpenVG 1.0 Accelerator achieved a 2.85 times faster performance in a tiger model.

Index Terms— OpenVG 1.0, 2D vector graphics, Graphics accelerator, Graphics pipeline.

I. INTRODUCTION

Recently, mobile devices need smooth, high-quality 2D graphics to enable high-quality user interfaces and ultra-readable text on small screens[1].

Most traditional 2D graphics are in the format of bitmap graphics that work efficiently with static contents at a consistent resolution. However, the storage requirement for animated bitmap graphics grow rapidly since each frame of animation must be stored as a separate bitmap. Also, if the same contents are displayed with different resolutions, an image filtering is required to blur sharp patterns such as text when the images are minified, or to create blocking artifacts when the images are magnified.

Vector graphics have two advantages: The file size tends to remain small, and the image can be scaled to any size without any degradation of the image quality. Since mobile devices usually do not have hard drives, and the screen size and even orientation varies a lot, vector graphics have strong advantages over bitmap graphics on mobile devices.

OpenVG is a royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphic libraries such as Flash and SVG. OpenVG is targeted primarily on handheld devices that require portable acceleration of high-quality vector graphics for compelling user interfaces and text on small screen devices - while enabling hardware acceleration to provide[2].

In this paper, we propose the hardware architecture to accelerate 2D Vector graphics process for mobile devices.

II. OpenVG Pipeline

An implementation of OpenVG may have an overall pipeline with 8 stages, as described in the OpenVG official specification. Since the implementers are not restricted to use the ideal pipeline mechanism, they can use any variations and/or even their own internal architectures. The only restriction is to provide the same result as the specification described. The overview of the OpenVG pipeline is represented in Figure 1.

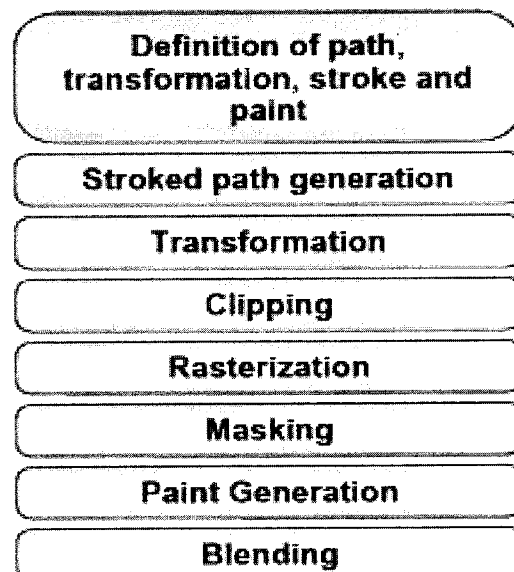


Fig. 1 OpenVG Pipeline

The first stage is a Definition of Path, Transformation, Stroke, and Paint. The application defines the path to be drawn, and sets any transformation, stroke, and paint parameters or leaves them at their default settings. When

Manuscript received March 12, 2008; revised July 18, 2008. Jae-Chang Kwak[1], Kwang-Yeob Lee[2],
[1]Dept. of Computer Science, Seokyeong University,
[2]Dept. of Computer Engineering, Seokyeong University
[2]Corresponding Author, Seo-kyeong Univ. Jeongneung 4-dong, Seongbuk-gu, Seoul, Korea

all parameters have been set, the application initiates the rendering process by calling vgDrawPath or vgDrawImage. If the path is to be both filled and stroked, the remainder of the pipeline is invoked twice in a serial fashion, first to fill and then to stroke the path.

The second stage is a Stroked path Generation. If the path is to be stroked, the stroke parameters are applied in the user coordinate system to generate a new path that describes the stroked geometry.

The third stage is a Transformation. The current path-user-to-surface transformation is applied to the geometry of the current path, producing drawing surface coordinates. For an image, the outline of the image is transformed using the image-user-to-surface transformation.

The fourth stage is a Rasterization. A coverage value is computed at pixels affected by the current path using a filtering process, and saved for use in the anti-aliasing step.

The fifth stage is a Clipping and Scissoring. Pixels not lying within the bounds of the drawing surface, and (if scissoring is enabled) within the union of the current set of scissor rectangles are not drawing. An application-specified alpha mask image is used to modify the coverage values generated by the previous stage.

Next is a Paint Generation stage. At each pixel of the drawing surface, the relevant current paint is used to define a color and an alpha value. For gradient and pattern paints, the paint-to-user transformation is concatenated with the path-user-to-surface transformation to define the paint transformation that will geometrically transform the paint.

The seventh stage is an Image stage. If an image is being drawn, an image color and alpha value is computed at each pixel by interpolating image values. The results are combined with the paint color and alpha values according to the current image drawing mode.

The last stage is a Blending and Anti-aliasing. At each pixel, the source color and alpha values from the preceding stage are converted into the destination color space and blended with the corresponding destination color and alpha values according to the current blending rule. The computed coverage value from stage 5 is used to interpolate between the blending and anti-aliasing.

III. Proposed Pipeline

The proposed pipeline of OpenVG is shown in Fig 2. The rasterizer stage contains clipping and scissoring units that are processing with coverage values.

Clipping doesn't generate edge that out of screen. So, it can reduce extra pipeline operation. Per pixel operation stage contains the steps such as Paint Generation, Blending, Masking, and Anti-aliasing.

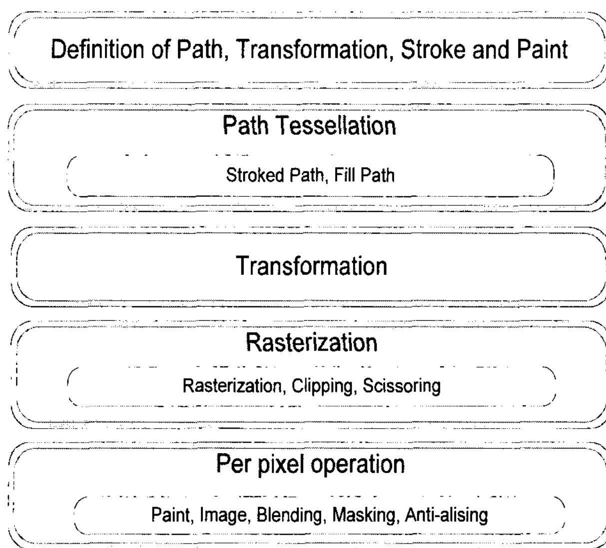


Fig. 2 Proposed Pipeline

3.1 Proposed Pipeline : Transformation

Transformation is processed by Affine transformation with matrix operation such as Translation, Scale, and Rotation. Transformation process needs 4 times floating point addition and multiplication for a coordinate change of 1 vertex. Equation 1 shows the this operations.

$$NV.X = V.X \bullet M[0][0] + V.Y \bullet M[0][1] + M[0][2]$$
$$NV.Y = V.X \bullet M[1][0] + V.Y \bullet M[1][1] + M[1][2]$$

* NV : Vertex coordinate after Transformation
* V : Vertex coordinate before Transformation
* M : Matrix for Affine Transformation.

Equation 1. Affine Transformation

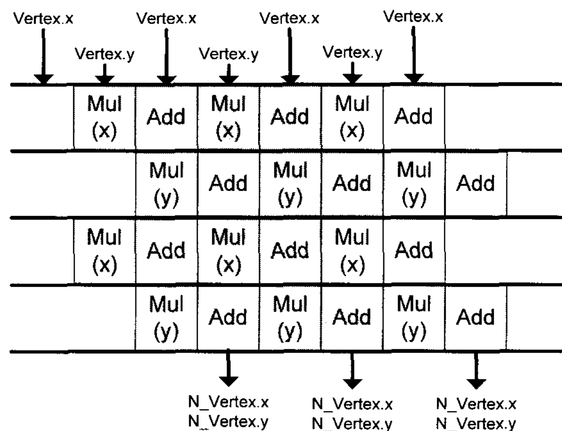


Fig. 3 Transformation Unit Architecture

Although Transformation uses 4 floating point adders and 4 floating point multiples, it requires 3 cycles execution time for the result because of the operation dependency.

In this paper, we propose the Transformation Unit Architecture that considerates the operation dependency.

It has 3 cycles execution time and uses 2 multipliers and 2 adders.

3.2 Proposed Pipeline : Rasterization

In case of the standard scan-line algorithm, it needs to generate Active Edge Table (AET) and to sort them in order of X coordinate while executing the scanline processing. Rasterizer uses the scan-line edge flag algorithm by Ackland et al.[5] with super sampling. The edges of the polygon are first plotted to a temporary 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 4 illustrates the filling operation with the edge-flag algorithm.

Sorting an array with AET is complex and it brings overhead with an additional memory operation. Proposed rasterizer is designed without sorting arrays with AET.

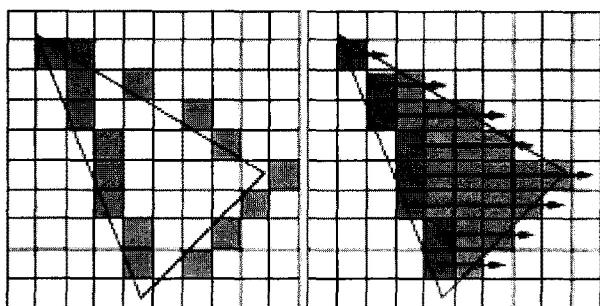


Fig. 4 Conventional Scanline Edge Flag Algorithms

A conventional edge-flag algorithms only supports the even-odd fill rule. If the application requires non-zero winding, the plain edge-flag is not enough, because it doesn't contain a direction information of the edge. In the 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.

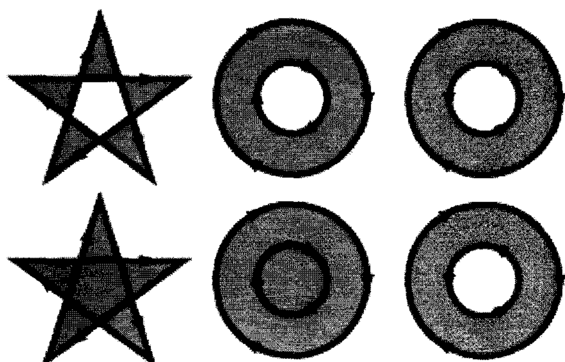


Fig. 5 Even-odd vs Non-zero fill-rule

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

In order to compute the winding count for Non-Zero fill rule, the winding count of the corresponding area should be calculated with the information of the area that the edge passes through. For this computation, a winding buffer whose size is a scanline is added. The winding buffer contains 8bits per fixel, and the value that is accumulated with winding of edges passing through the corresponding fixel. Fig. 5 shows the activity of the winding buffer.

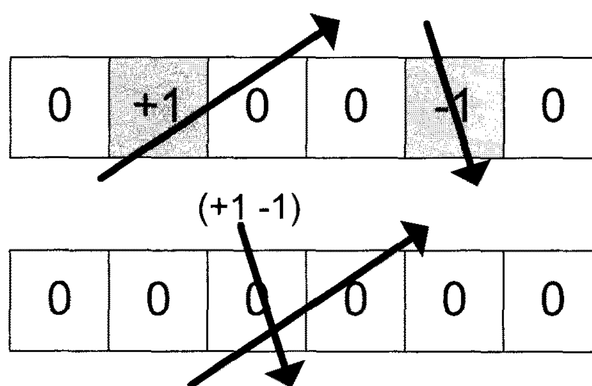


Fig. 6 Winding Buffer

The flag value is recorded at the mask buffer. The recorded location is corresponding to the overlapping point of the edge and the scanline with the rendering by the extension of conventional scanline edge-flag algorithm. The edge direction value is recorded at the same position of winding buffer.

This method can dramatically reduce the overhead of memory computation, because edge data are not needed to be fetched for the calculation of the winding count of each fixel. But additional memory spaces are needed to store winding counts

To support Anti-Aiasing, Masking Buffer and Winding Buffer must be added in proportion to the number of sampling. Our proposed 2D Vector Graphics Accelerator supports two sampling methods, four or eight samplings per fixel. The size of Mask Buffer and Winding Buffer should be as follows:

Mask-Buffer : Scanline size * 8bits
Winding-Buffer : Scanline size * 8bits * 8

The reason to define each bit of Mask Buffer as 8 bits is the need to mark 8 bits per scanline in order to perform maximum 8 samplings per scanline. Fig. 7 shows this procedure as a unit of 4 bits

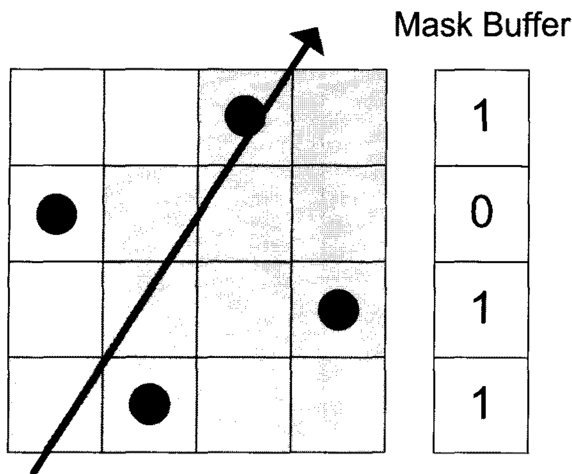


Fig. 7 Super sampling with Mask buffer

The Coverage Value of each pixel can be calculated easily by Mask Buffer of the pixel. In the next pixel pipeline step, additional computations for Anti-Aliasing is not needed by using precomputed Coverage Values. The above algorithm can be applied to Anti-Aliasing of the Non-zero fill rule with small amendments.

3.3 Proposed Pipeline : Paint Generation

Processing of generating paint produces a gradient paint which follows the Paint-mode.

In traditional method, to compute per pixel color of gradient paint, it computes a gradient offset value. This computed color of final pixel is used as interpolated two colors.

Proposed paint generation unit uses a LUT method, so it does not execute color interpolation which needs to be calculated every time. LUT is generated when the input receives a range price of color to set the first gradient color, and the color is calculated using the generated LUT after by a process.

IV. Verification

As a result of analysis of the operation performance from tiger sample image, we can find that it frequently uses floating point addition and multiplication, square root, and division. Because it often uses a mathematics operation in tessellation and paint steps, through them, we need the improvement of speed with H/W realization to realize OpenVG with floating point.

Verification is performed to measure both functionality and performance. Functionality verification checks whether the functionalities, suggested by OpenVG specification, works correctly. Fig. 8 shows the result of functional verification of Dash pattern and Cap/Join style functions in the image rendering at Stroke path step.

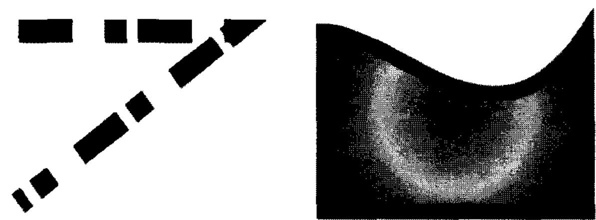


Fig. 8 Dash pattern & Cap/Join style

Fig. 9 Tiger image with Scissoring

Fig. 9 shows the result of rendering of Tiger sample image. Scissoring function is applied to draw the image only at the specified space. Two Scissoring rectangle spaces are assigned, and the rendering image is displayed at the inside of the rectangle.

Table 1 illustrates the performing time to Tessellate path that viewed image on Fig. 10 and compared with OpenVG reference. Table 2 presents the time to generate paint colors for the Gradient paint-mode.

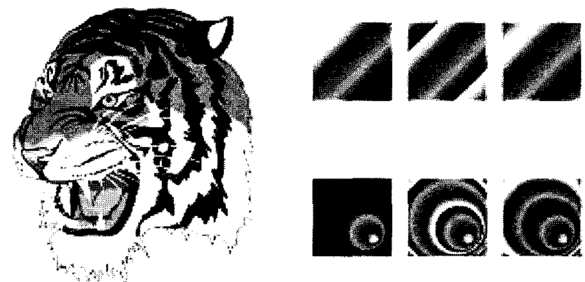


Fig. 10 Tiger & Gradient Image

Table 1. Image Rendering Time of Tiger & Gradient paint

	Reference	Proposed
Tiger	593ms	208ms
Radial Gradient	63ms	21ms
Linear Gradient	58ms	27ms

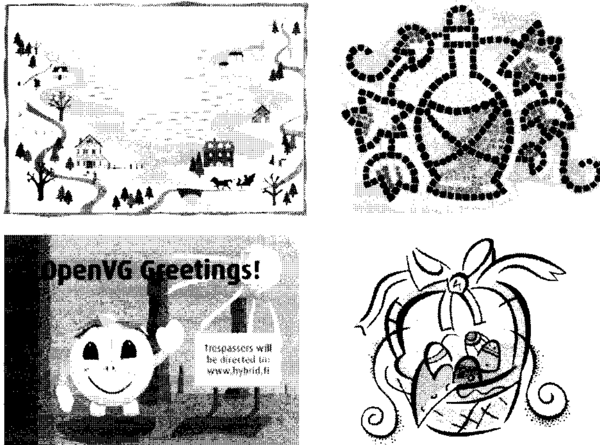


Fig. 11 Verification Images
(Snow, bottle, dude, flower)

Fig. 11 shows OpenVG images, used to compare Rendering speed with the reference image. Table 2 illustrates the formance result of speed comparison.

Table 2. Image Rendering Time of Verification Images

	Reference	Proposed
snow	362ms	132ms
bottle	412ms	177ms
dude	386ms	156ms
flower	397ms	188ms

V. Conclusion

In this paper , we propose new OpenVG pipeline and algorithm which is composed of 2D vector graphics pipeline, and configured OpenVG pipeline architecture. For mobile environment, we uses floating point data type which is useful in reducing the additional cost in realization of software and hardware. The proposed new pipeline fits for hardware realization grouped by functions, or operations.

The project is verified with accuracy test of movements and functions by comparing our developed OpenVG with Tiger Sample Image offerd by Khronos group. Through the verification program, we verified and realized several functions.

ACKNOWLEDGMENT

This work was supported by "Nano IP/SoC Innovative Promotion Group" and "ETRI SoC Industry Promotion Center".

REFERENCES

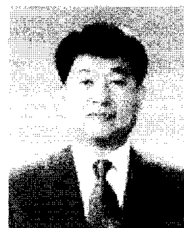
[1] Kari Pulli, "New APIs for Mobile Graphics", Pr oceedings of SPIE - The International Soci ety f or Optical Engineering Vol. 6074, art, no. 60740 1, 2006

[2] Khronos Group Inc. "OpenVG Specification Vers ion 1.0.1" http://www.khronos.org/openv g/, Janu ary 2007

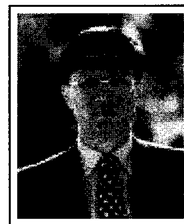
[3] ARM "Fixed Point Arithmetic on the ARM" , A pplication Note 33, ARM, September 1996

[4] Kiia Killio "Scanline edge-flag algorithm for anti aliasing" EG UK Theory and Practice of Comput er Graphics 2007

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



Jae Chang Kwak
received the B.S degree from Yonsei University in 1983. He received the M.S and Ph.D degrees in computer science from the University of Iowa in 1989 and 1993, respectively.
He is currently an Professor of Computer Science at Seokyeong University. His main interests are Network Traffic control, Realtime Scheduling, Embedded System, Mobile Graphics System.



Kwang Yeob Lee
studied electronics engineering at Sogang University and Yonsei University from 1979 to 1987. In 1994 he received the Ph.D from the Yonsei University. From 1989 to 1995, he was with Hyundai Electronics as a designer of System LSI. During that time, he was responsible for the design of microcontroller. In 1995, he joined the Department of Computer Engineering , Seokyeong University. His research interests include Embedded System, Mobile 3D Graphics Accelerator, SoC Design.