The 23rd International Technical Conference on Circuits/Systems,
Computers and Communications (ITC-CSCC 2008)

A Design of OpenVG 2D Vector Graphics Accelerator for a Mobile Device

Jeong-Hun Park*, Kwang-Yeob Lee*, Jae-Chang Kwak**
* Department of Computer Engineering, Seokyeong University
**Department of Computer Science, Seokyeong University
16-1, Jeongneung-Dong, Seongbuk-Gu, Seoul 136-704, Korea
Tel: +82-2-940-7240, Fax: +82-2-940-7240
E-mail: seiclub@skuniv.ac.kr
Keyword : 2D Graphics, OpenVG, Vector Graphics

1. Introduction

Recently, mobile devices need smooth and 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. The storage requirements for animated bitmap graphics grow rapidly, since each frame of animation must be stored as a separate bitmap. If the same contents are displayed on screens with different resolutions, the images need to be filtered, which blur sharp patterns such as text when the images are minified and created the 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 major 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 a mobile device.

2. Basic 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.

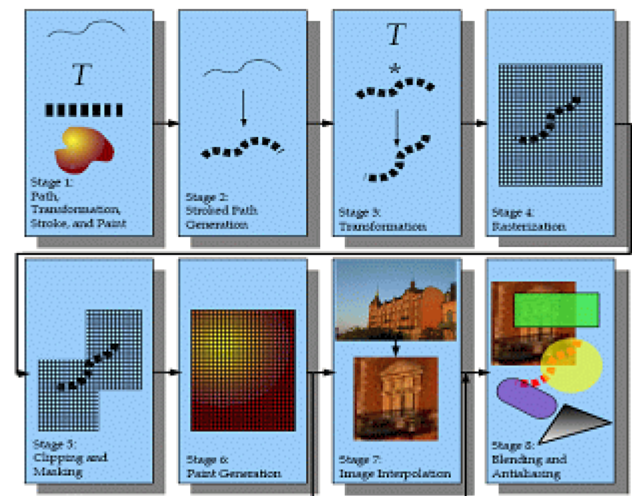

Figure 1. OpenVG Pipeline

Stage 1: 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 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.

Stage 2: 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.

Stage 3: 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.

Stage 4: Rasterization

A coverage value is computed at pixels affected by the current path using a filtering process, and saved for use in the antialiasing step.

Stage 5: Clipping and Masking

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.

Stage 6: Paint Generation

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.

Stage 7: Image Interpolation

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.

Stage 8: 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.

3. A proposed pipeline

A proposed pipeline of OpenVG is shown in Fig 2. The rasterizer stage contains clliping and scissoring units that are processing with coverage values. Clipping doesn't generate edges that are at the out of screen. So, it can reduce extra pipeline operations. Per pixel operation stage contains the steps such as Paint Generation, Blending, Masking, and Antialiasing.

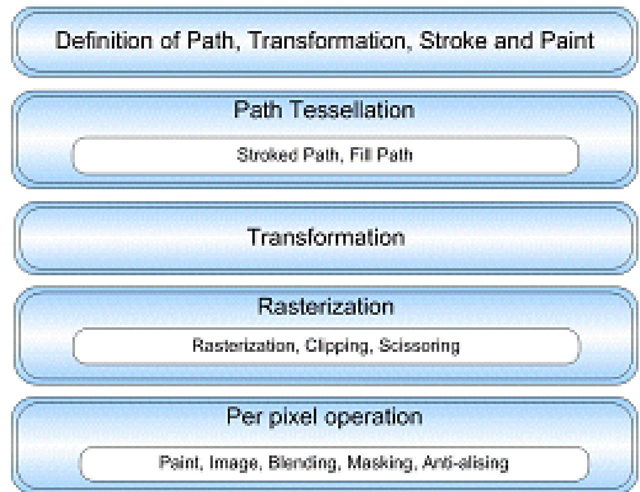

Figure 2. A proposed Pipeline

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

NV.X = V.X * M[0][0] + V.Y * M[0][1] + M[0][2]
NV.Y = V.X * M[1][0] + V.Y * 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

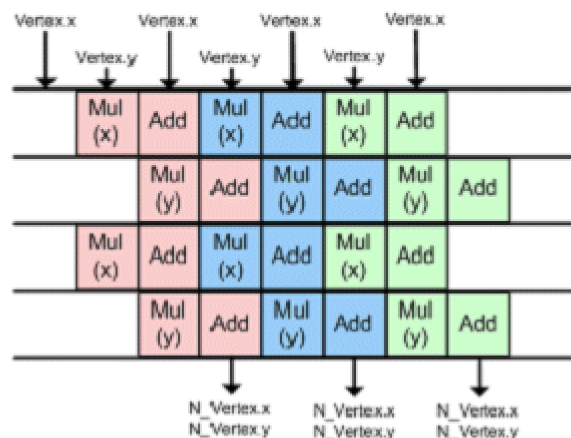

Figure 3. Transformation Unit Architecture

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 Transformation Unit Architecture that considerates the operation dependency. It has 3 cycles excution time and uses 2 multipliers and 2 adders.

The standard scan-line algorithm generates Active Edge Table (AET) and sort them in order of X coordinate while excuting 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 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 an 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 additional memory operation. The proposed rasterizer is desinged without sorting arrays with AET.

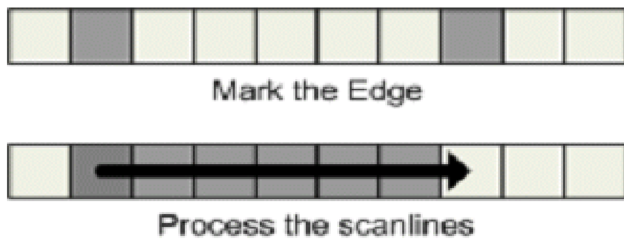

Figure 4. Scanline Edge Flag Algorithms

A conventional edge-flag algorithms only supports the even-odd fill rule. If an application requires non-zero winding, plain edge-flag is not enough, because it doesn't contain 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, and 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, and if it is zero, the pixel is empty

The algorithm can be extended rather easily to support the non-zero winding fill rule. However, this is done at the cost of memory usage. To support the non-zero fill rule, Rasterizer has a mask-buffer (same size of scanline) and a winding-buffer(same number of sampling)

.

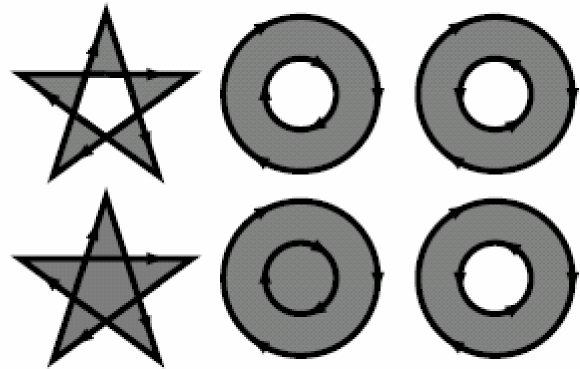

Figure 5. Even-odd vs Non-zero fill-rule

An Anti-Aiasing uses the coverage value which was produced in a processing of rasterization, so it does not need an extra processing. Processing of generating paint produces a gradient paint which follows the Paint-mode. In traditional method, a gradient offset value is computed to compute per pixel color of gradient paint, and this computed color of final pixel is used as an interpolated two color. The proposed paint generation unit using a LUT method so it does not execute color interpolation which is needed to calculate 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 generated LUT by a process..

4. Verification

As a result of analysis of the operation performance from tiger sample image, we 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, we can achieve the improvement of speed with H/W realization to realize OpenVG with floating point.

Table 1 illustrates the performing time to Tessellate path that viewed image on Figure 6 and compared with OpenVG reference. Table 2 presents the time to generating paint color for Gradient paint-mode. Table 3 shows the total image rendering times

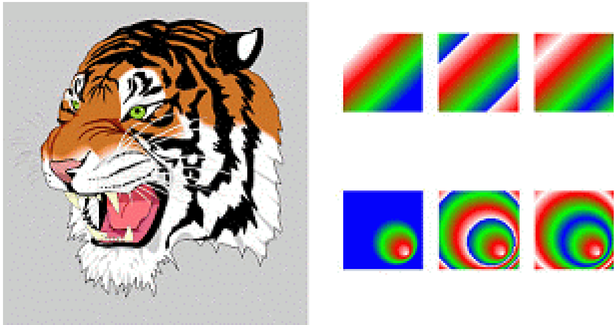

Figure 6. Tiger & Gradient Image

	Reference	Proposed
Tiger	167ms	12ms
Radial Gradient	2ms	1ms
Linear Gradient	2ms	1ms

Table 1. Tiger & Gradient Image Tessellation Time

		Reference	Proposed
Radial Gradient	Pad	36ms	3ms
	Repeat	37ms	4ms
	Reflect	37ms	3ms
Linear Gradient	Pad	21ms	3ms
	Repeat	22ms	4ms
	Reflect	21ms	3ms

Table 2. Gradient paint generation time

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

Table 3. Total Image Rendering Time

5. Conclusion

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

The project are verified with the accuracy test of movements and functions to compare our developed OpenVG with Tiger Sample Image offerd by Khronos group. We verified and realized several function to perform in OpenVG through the verification program.

References

[1] Kari Pulli, *"New APIs for Mobile Graphics",* Proceedings of SPIE - The International Soci ety for Optical Engineering Vol. 6074, art, no. 607401, 2006

[2] Khronos Group Inc. *"OpenVG Specification Version 1.0.1"* http://www.khronos.org/openv g/, January 2007

[3] ARM "*Fixed Point Arithmetic on the ARM*" , Application Note 33, ARM, September 1996

[4] Kiia Killio "*Scanline edge-flag algorithm for antialiasing*“ EG UK Theory and Practice of Computer Graphics 2007

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