Implementation of OpenVG 1.0 using OpenGL ES

Aekyung Oh, Hyunchan Sung, Hwanyong Lee
HUONE INC.
Doge Bldg 969-9 Dongchun Daegu
702-250, Korea
82 53 325 4956
{oak, hcsung, hylee}@hu1.com

Kujin Kim
Dept, of Computer Engineering,
Kyungpook Nat'l Univ.,
Daegu 702-701, Korea
82 53 953 7573
kujinkim@yahoo.com

Nakhoon Baek
School of EECS,
Kyungpook Nat'l Univ.,
Daegu 702-701, Korea
82 53 950 6379
oceancru@gmail.com

ABSTRACT
OpenVG 1.0 is a 2D vector graphics standard and its API (Application Programming Interface) was released by the Khronos Group. In this paper, we introduce our OpenVG 1.0 implementation, accelerated by OpenGL ES 1.x hardware. Our implementation is an efficient and cost-effective way of accelerating OpenVG, fully utilizing the existing hardware in current embedded systems. Conclusively, our OpenVG implementation shows dramatically outstanding performance with low power consumption.

Categories and Subject Descriptors
I.3.0 [**Computer Graphics**]: *2D graphics engine for embedded system using 3D hardware accelerator chip*

General Terms
Algorithms, Performance.

Keywords
OpenVG, Vector Graphics API, Vector Graphics Engine

1. INTRODUCTION
In the area of embedded systems, we have steadily increasing needs for 2D vector graphics solutions. In July 2005, Khronos Group released a 2D vector graphics standard, OpenVG 1.0[1]. In spite of its elegant features, this API is not yet widely used, since its software implementations did not show acceptable performances while hardware implementations invoke cost problems. We present a cost effective way of accelerating OpenVG API, using the existing hardware solutions for OpenGL ES[2], which is a 3D graphics standard mainly used for embedded systems.

2. Related Technique

2.1 OpenVG
OpenVG is a royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG.

An Ideal graphics engine for OpenVG is constructed by a pipeline as shown in [Figure 1]. Path data are handled sequentially through the stages 1 to 6 and 8, while image data are handled through all the stages 1 to 8.

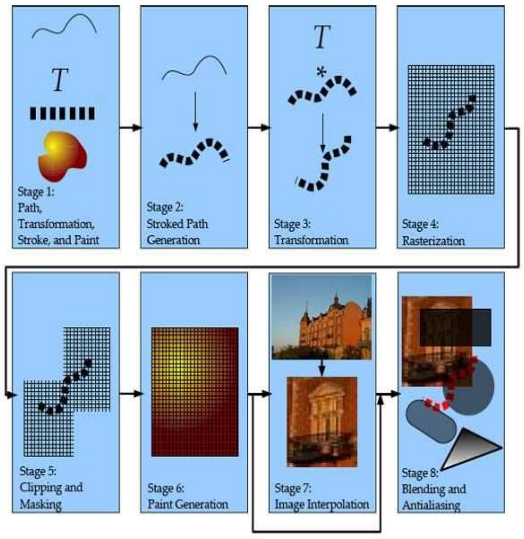

Figure 1. OpenVG Rendering Pipeline

Each pipeline stage of OpenVG 1.0 plays a role as follows:

Stage 1 (Define Drawing Parameter & Coordinate)
Users input coordinates, shapes that they want and then decide drawing parameters. For instance, the parameters would be stroke line width, stroke paint color, fill paint color, transform matrices, and so on.

Stage 2 (Stroked Path Generation)
In stroked path generation stage, we offset the input path by using the fixed stroke width.

Stage 3 (Transformation)

Transformation stage transforms the geometric information of paths defined in user coordinates system to that in screen coordinates system. The coordinate transformations are performed by multiplying a 3x3 matrices and vectors.

Stage 4 (Rasterization)
In rasterization stage, we calculate coverage values of pixels that are affected by current path. The coverage value is stored and then used later in antialiasing stage (Stage 8). This is the first stage that converts vector data into bitmap data.

Stage 5 (Clipping & Masking)
This stage performs clipping and masking to display the specified area of the screen only.

Stage 6 (Paint Generation)
This stage decides the color that will be painted to the result area generated in stage 5. Consequently, this stage performs fill & stroke operations with a single fill color, a single stroke color, gradations and pattern images.

Stage 7 (Image Interpolation)
If users draw images, this stage calculates each pixel value of the interpolating image with inverse of image matrix.

Stage 8 (Blending & Antialiasing)
This stage generates antialiased bitmap data with coverage and pixel values that were stored from previous stages, and then blends the current bitmap and the previous bitmap for the overlapping operation. Methods of the blending are Porter-Duff Blending modes, multiply, screen, darken, lighten, additive etc.

2.2 OpenGL ES

OpenGL ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems. It consists of well-defined subsets of desktop OpenGL [3], creating a flexible and powerful low-level interface between software and graphics acceleration. We may consider the pipeline of OpenGL ES and that of OpenGL are identical.

General pipeline architecture of OpenGL ES (or OpenGL) is shown in [Figure 2].

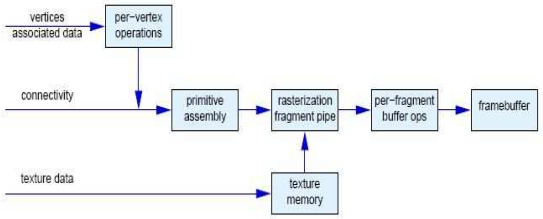

Figure 2. OpenGL ES Rendering Pipeline

Each pipeline stage of OpenGL ES plays a role as follows:

Per vertex Operations
This stage performs a transformation for user's input vertices and calculations for lighting.
Primitive assembly
This stage combines a set of vertices to form a primitive.
Rasterization
This stage converts graphics primitives into fragments and performs the fragment pipeline.
Per fragment Operations
This stage performs image texturing, color summation and fog adjusting.
Buffer Operations
This stage performs a depth/stencil test and alpha blending.

OpenVG pipeline is similar to OpenGL ES pipeline, but it requires follow the process to suitably covert the data that used in OpenVG API to those for OpenGL ES API. This process is necessary since the data types in both APIs are different.

3. Architecture

OpenVG was implemented with OpenGL ES accelerator, and its overall architecture is shown in [Figure 3]. OpenVG API in application areas can accelerate 2D objects rendering with OpenGL ES accelerator, and also display both 2D and 3D objects with OpenVG API and OpenGL ES API at the same time.

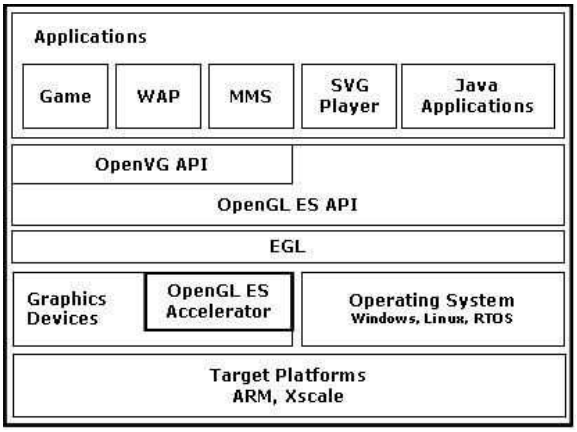

Figure 3. Architecture overview

4. Implementation

Fundamentally, our work is implementing OpenVG features by using OpenGL ES functionalities. Thus, our basic strategy is mapping OpenGL ES functionalities to each of OpenVG features, as shown in Table 1.
We also have some important implementation issues and will show them. First, OpenGL ES only works with lines and triangles, while OpenVG needs general polygon filling operations due to stroke paths of curves and arcs. Consequently, starting from traditional sweep-line algorithms [4], we developed a specialized polygon triangulation algorithm that supports all OpenVG filling rules. Since complex scenes often require heavy computation efforts for this triangulation step, we use a cache mechanism for managing the triangulation results.

Paint (solid fill, linear and radial gradation, and pattern fill) and image functions of OpenVG are rendered with OpenGL ES texture functions and their results are stored in the OpenGL ES texture memory area for more efficient computation. Alpha masking is implemented with multi-texturing features of OpenGL ES. Due to these texturing operations, our OpenVG

implementation may require at most 2 textures for simple paths, and at most 3 textures for image, paint and masking operations.

Our OpenVG engine is fully verified with ATI Radeon X600 and nVIDIA GoForce 4800, and now ready for commercial services to support new user interface paradigms on major-brand mobile phones.

Table 1. OpenVG implementation with the functionality of OpenGL ES 1.x

OpenVG 1.0 feature	How to Implement it on OpenGL ES
Paths (Move, Line, Curve, Arc)	with Tessellation
Fill & stroked (Solid, Dashing)	with Tessellation and depth-buffer
Fill rule (Nonzero, Even-odd)	Modify Tessellation Algorithm
Paints(Color, Linear & Radial gradient, Pattern)	With OpenGL ES functions
Color ramp spread modes (Pad, Repeat, Reflect)	Making new texture
Pattern tiling modes (Fill, Pad, Repeat, Reflect)	Making new texture
Images (Normal, Multiply, Stencil)	with Textures Function (Stencil is not implemented)
Filters	Software implementation
Rendering & image Quality	Not implemented (HW dependent)
Matrices	with OpenGL ES matrices
Scissoring	with Stencil buffer
Alpha Masking (Clear, Fill, Set, Union, Intersect, Subtract)	with Texture functions
Blend modes (Over, In, Multiply, Screen, Additive, Darken, Lighten)	With OpenGL ES Blend functions(Darken& Lighten are not implemented)

5. Performance

Performance result of the implementation that studied in this paper with nVIDIA GoForce 4800(CPU: ARM9 266MHz) is shown in table 2. In table 2, the row with H/W in the first column shows the performance of our work. The row with S/W in the first column shows the performance of the software engine (HUONE's AlexVG [5]) that is only a software implementation without hardware acceleration.

Table 2. Performance (FPS)

Demos	Tiger	Map Service	Flakes of flowers	Child images
Images (240x320)				
DESC.	Fill & stroked paths	Map data paths	Drawing paths & image	Drawing child images
H/W	6.15	11.3	12.8	10.0
S/W	1.8	2.2	4.3	0.4

In most cases OpenVG engine accelerated by OpenGL ES Hardware is much faster than OpenVG engine (HUONE's AlexVG), which is implemented in fully software.

6. Conclusion

In this work, we implemented an OpenVG 1.0 renderer on the OpenGL ES hardware. Our implementation is quite cost-effective in comparison with the case when developing pure-hardware OpenVG solutions. Moreover, our implementation shows a dramatically improved performance comparing with previous software implementations. Our work also shows a new application area of the OpenGL ES hardware.

7. ACKNOWLEDGMENTS

This work is financially supported by the Ministry of Education and Human Resources Development (MOE), the Ministry of Commerce, Industry and Energy (MOCIE) and the Ministry of Labor (MOLAB) through the fostering project of the Industrial-Academic Cooperation Centered University.

8. REFERENCES

[1] Khronos Group: OpenVG Specification 1.0. (2005) [http://www.khronos.org/openvg]

[2] Khronos Group: OpenGL ES Specification 1.1 (2004) [http://www.khronos.org/opengles]

[3] Khronos Group: OpenGL Specification 2.1 (2006) [http://www.khronos.org/openg]

[4] M. Berg, et. al., *Computational Geometry*, Springer, (1997).

[5] AlexVG Forum: AlexVG 1.0 (2006) [http:// www.alexvg.com]