Vector Graphic Reference Implementation for Embedded System

Sang-Yun Lee[1] and Byung-Uk Choi[2]

[1]Dept. of Electronical Telecommunication Engineering, Hanyang University, Seoul, Korea
syllee@etri.re.kr
[2]Division of Information and Communications, Hanyang University, Seoul, Korea
buchoi@hanyang.ac.kr

Abstract. We propose the reference implementation with software rendering of OpenVG for the scalable vector graphic hardware acceleration, which the Khronos group standardizes. We present the design scheme that enables EGL and OpenVG to be ported easily in an embedded environment. Moreover, we describe the background of selection of an algorithm, and the mathematical function adopted for the performance improvement, and we propose the optimum rendering method. We present displaying of vector image on a screen through the OpenVG implemented using software rendering method. And, we present the test result of the CTS which is compatibility test tool. And we show the performance comparison against the Hybrid corp.'s reference implementation.

Keywords: OpenVG, EGL, Scalable Vector Graphic, Embedded System, Software Rendering

1 Introduction

Recently, the demand for the applications using the vector graphics technology has increased [1]. Particularly, in areas such as SVG viewer, hand-held guidance service, E-Book reader, game, scalable user interface, and etc, the vector graphics technology is widely applied. 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 [2].

Currently in development, OpenVG is targeted primarily at 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 fluidly interactive performance at very low power levels. OpenVG is the standard constituted by the Khronos group. And the version 1.0 was released at July 2005 for the first time [3].

When the standard needs to be verified, or when the OpenVG application needs to be operated through an emulator in advance, or when there is a no hardware supporting OpenVG, it is necessary to have the reference implementation (RI) operating in the software rendering mode. Additionally, it takes much time until the special-

purpose hardware supporting OpenVG is produced. The RI also can reduce the cost. Besides, as embedded devices and CPU's performance is improved, the possibility of being replaced with the software rendering is high.

In this paper, we propose the OpenVG reference implementation which can be easily ported to the various embedded devices by using the software rendering method. And, we show that our RI is more excellent than the existing RI in the performance aspect.

2 Design of OpenVG and EGL Engine

2.1 System Architecture

The system architecture of the OpenVG RI proposed in this paper is shown in Fig. 1.

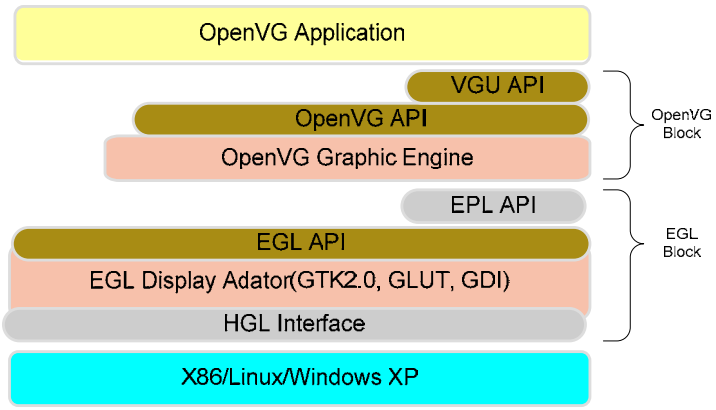


Fig. 1. The system architecture

The OpenVG RI is composed of Embedded Graphics Library (EGL) block and OpenVG block. EGL is an interface between rendering APIs such as OpenGL|ES or OpenVG (referred to collectively as client APIs) and an underlying native platform window system [4]. EGL provides mechanisms for creating rendering surfaces onto which client APIs can draw, creating graphics contexts for client APIs, and synchronizing drawing by client APIs as well as native platform rendering APIs [5]. We designed so that, through the EGL Display Adapter, the client API could access the windowing system of the native platform.

In the EGL standard, the Embedded Platform Library (EPL) API is not included. However, it is necessary in order to implement client API, and must be ported according to a system. The API includes, for example, the functions of returning the frame buffer from Surface, the memory allocation, memory releasing, and etc. Hardware Graphic Library (HGL) interfaces performs the function of connecting EGL to the na-

tive graphics system. EGL is itself the standard which is made to abstract the hardware system. However, it is necessary to have the separate porting layer like HGL so that EGL can be ported to the different native platform window systems through the minimum overhead. The OpenVG API is 2D vector graphics library and the VGU API is 2D vector graphics utility API of the high level. The OpenVG graphic engine provides the core functions that the OpenVG API and the VGU API need.

2.2 Structure of the EGL Engine

The block structure of the EGL engine is shown in Fig. 2.

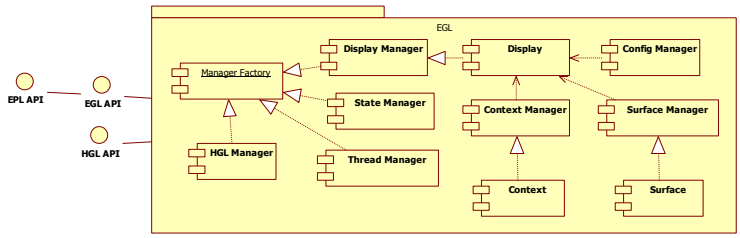


Fig. 2. Structure of the EGL Engine

The Display Manager creates and manages the display object which takes charge of displaying graphics. The State Manager stores the error value generated when the functions executed. The Thread Manager provides the functions for avoiding the race condition in which several processes or the threads try to access EGL at the same time. As to these three modules, however, only one object can be generated. The Manager Factory enables them to have the uniqueness in three modules.

2.3 Structure of the OpenVG Engine

The block structure of the OpenVG engine is shown in Fig. 3. The OpenVG engine provides the OpenVG API which applications can use and the Context Sync API which EGL can use. The Context Sync API provides synchronization between the VGContext generated in the OpenVG internally and the Context generated in EGL. The elements that need synchronization include creation and termination of a context, the current setup status, and etc. The VG State Manager stores and manages the error value generated during the execution of OpenVG. The VG Context Manager, the Paint Manager, the Image Manager, and the Path Manager create and manage the VG Context object, the Paint object, the Image object, and the Path object respectively. The VG Manager Factory performs the role of guaranteeing that these objects operate with singleton. The Rasterizer performs the function of drawing on the

frame buffer provided by the surface of EGL with data combined of Path, Image, and Paint information.

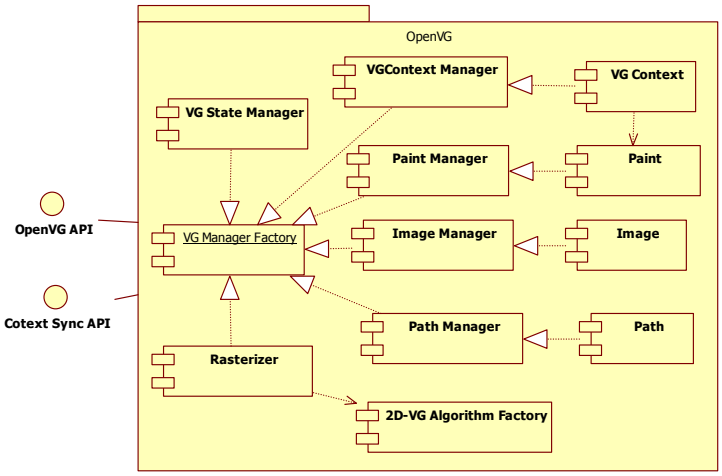


Fig. 3. The Structure of the OpenVG Engine

2.4 Requirement for Designing the OpenVG/EGL Engine

OpenVG can operate not only on a desktop personal computer but also on a server. But it was developed to be mainly used in the embedded devices. In an embedded environment, there are always porting issues, because of wide variety of not only the hardware, but also the platforms and software. Therefore, we must consider porting issues at as early as the architecture design phase, if we want to easily port once developed OpenVG to the various embedded devices. That is, the porting layer must exist so that the part to be modified according to the environmental change can be minimized. In addition, EGL and OpenVG must be loosely coupled.

Generally, in an embedded environment, the performance of a CPU is lower and the size of the memory is restrictive. Therefore, algorithms must be selected in such a way that the selected algorithms produce optimum performance and at the same time use as low memory and power as possible.

3 Novel Features of OpenVG Reference Implementation

3.1 Mathematical Function

In the operation process of drawing each graphic object of OpenVG, the use of the mathematical function is frequent. The method of calculating the mathematical function is classified into two ways. Firstly, it is the method of referring to the table value having the value calculated in advance. The second is the method of calculating the Taylor series of the finite order [6]. The former case consumes big amount of memory, whereas the latter takes longer time to execute.

The Hybrid RI [7] adopted the later case. And as a result, it induced the performance degradation of the mathematics functional operation [8]. However, we adopted the table-look-up method and sought the performance improvement.

3.2 Sort Algorithm

In the OpenVG, sorting is used in the tessellation based rendering algorithm. That is, the vertex passing the scan line is arranged to the abscissa order. Or there is case where it arranges several scissoring rectangles. In the Hybrid RI, the bubble sort algorithm was adopted. But we adopted the merge sort algorithm, in this paper. The merge sort has a complexity of $O(N \log N)$, whereas the bubble sort has a complexity of $O(N^2)$.

3.3 Improved Raster Rendering Algorithm

A rendering refers to the operation of drawing the vector graphics object in the display [9]. There are the vector rendering mode in which the vector graphics object is drawn every time, and the raster rendering mode in which objects are drawn by calculating the color of each pixels of an image [10], [11], [12].

The raster rendering has an advantage in comparison with the vector rendering in the various aspect. Firstly, the raster rendering has a lower complexity than the vector rendering according to increasing of the number or area of Path. Secondly, the calculation for applying the Fill Rule is made altogether in the vertex drawing step. Thirdly, the mathematical calculation for vertex drawing is unnecessary because Vertex is not directly drawn. But it spends much time, because this method calculates all the parts which are not in fact displayed in a screen [13].

In this paper, we propose the improved rendering algorithm in order to solve this problem. Fig. 4 shows the improved raster rendering algorithm proposed in this paper.

The procedure of the proposed rendering algorithm is as followings: (1) If the area overlapping with the path bound is discovered for each scissoring rectangle, the pixel color is calculated for the corresponding part; (2) If the area overlapping is not discovered, the same operation is performed for the next scissoring rectangle; (3) If the vertex intersecting with the corresponding scan line is not discovered, the scan line is moved to the next line. That is, by remarkably reducing the area visited in order to calculate the pixel value, the proposed method can display the vector graphics faster than the existing method.

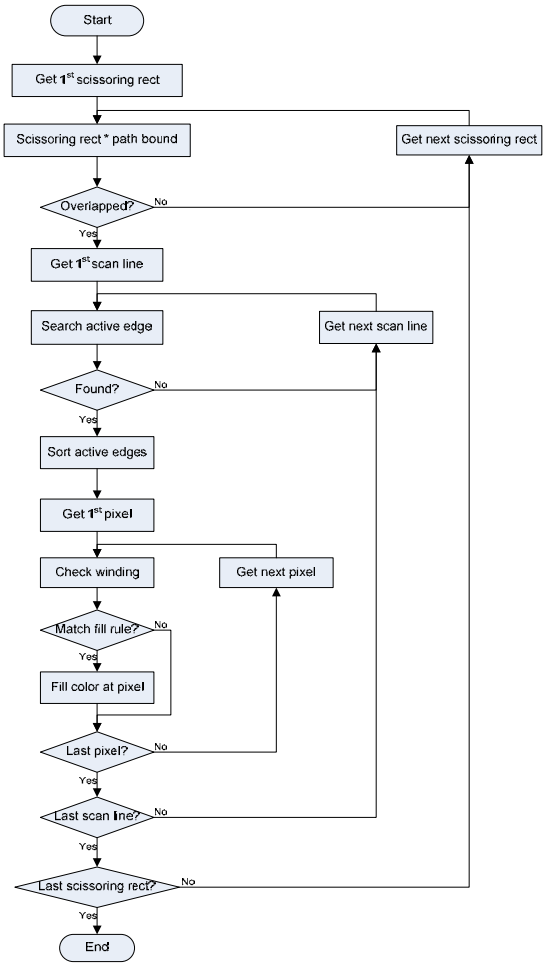


Fig. 4. The proposed Rendering Algorithm

4 The Design Point for the Embedded Environment

4.1 The Coherence of OpenVG and EGL

The Hybrid RI shares data structure between the objects which EGL and OpenVG create. Therefore, it has an effect on the other block if one block is modified among EGL or the OpenVG block. For example, the waste of resources occurs, because the object for OpenVG is also generated within EGL when the data structure of EGL is expanded to support OpenGL|ES and is used, although we develop only the OpenGL|ES application.

In this paper, we separated data structure of OpenVG and EGL and concealed each data structure and status information, in order to resolve these problems. Moreover, we designed so that OpenVG could use the function of EGL engine through the EGL API or the EPL API call.

4.2 The Language Dependency

The Hybrid RI was implemented with C++ language. C++ language is very powerful in the desktop environment, but it has many problems to be used in the embedded environment. Firstly, the speed of executing the inheritance or the virtual function is slow. Secondly, some compilers do not completely support C++ language. Therefore, we adopted C language that can be easily ported to an embedded device and the execution speed is fast. We defined function pointers in C structure that processes and manipulates the information, in order to provide for object-oriented concept supported easily in C++ language.

4.3 The Singleton Pattern Design

There is an object in which several copy creations are not allowed among an object. Each Factory in the EGL block, and the context Manager, state Manager, image Manager, and algorithm Factory in the OpenVG block are those objects. We introduced the singleton pattern for the guaranteed uniqueness of an object. If we design without the singleton pattern, each module has to recognize this object and avoid creating them. Or, these objects have to be registered in the global variables storage and used. However, existing code has to be modified to be ported to other platforms, because it is different in the structure of the global variables storage according to platforms.

5 Implementation and Experimental Results

5.1 Implementation

We verified whether our design method could be easily adapted to the embedded environment by implementing EGL and OpenVG and porting them to the various environments. We implemented EGL and OpenVG based on the Windows XP at first. And then, we modified the porting layer and could easily port it to the Linux and the WIPI platform [14]. At first, we implemented EGL for the Windows XP by using the GDI (Graphic Device Interface) in order to access the native windowing system, and then we ported it with the OpenGL Utility Toolkit (GLUT) for the performance comparison with Hybrid RI. There is the advantage of reducing the code amendment too, when it is ported to the Linux if it uses GLUT.

Fig. 5 shows vector graphics displayed on a screen through the proposed RI. Fig. 5(a) shows the image seen in the Tic-Tac-Toe game. Fig. 5(b) shows the tiger image that is the representative image of the vector graphics field. The Tiger image is comprised of 305 paths. Fig. 5(c) shows the tiger image rendered on a celluar phone.

(a) Tic-Tac-Toe on PC (b) tiger on PC (c) tiger on handset

Fig. 5. Example of vector graphic displayed using the proposed RI.

(a) 25% (b) 50% (c) 75% (d) 100%

Fig. 6. The process of drawing tiger

Fig. 6 shows the process of the tiger image being displayed with vector graphics. It shows when of 305 path, 25%, 50%, 75%, and 100% of the path are drawn.

5.2 CTS Test Result

Table 1. CTS test result

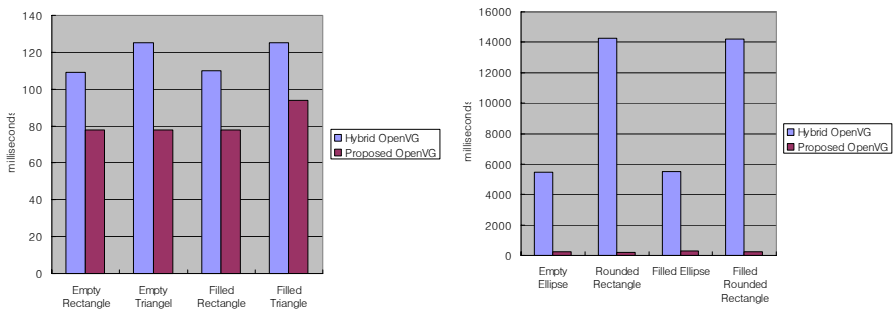
Item	No. of test	Success	Fail	Success Rate(%)
Parameter	10	8	2	80.0
Matrix	11	11	0	100.0
Clearing	3	3	0	100.0
Scissoring	5	4	1	80.0
Masking	2	2	0	100.0
Path	48	37	11	77.1
Image	10	6	4	60.0
Paint	10	5	5	50.0
Image Filter	3	2	1	66.7
VGU	12	5	7	41.7
Total	*114*	*83*	*31*	*72.81*

We performed test through the Conformance Test Suites (CTS) 1.0.0, the OpenVG compatibility test tool that the Khronos group distributes [15], in order to verify how our implementing RI adhered to the standard specification. Consequently, the success rate was 73% approximately. Table 1 shows the CTS success rate according to the test item in detail.

Among the items that CTS reported as failure, there were items that correctly rendered resulting image cannot be differentiated from our reference-generated image visually. This was the case where the pixel value was off by one pixel. Moreover, there was a case where the already passed test image failed when a code was modified in order to pass another failing case. This is grasped that the test image of CTS is not yet stabilized.

5. 3 Performance Evaluation

We developed performance measure program called Vgperf for 2D vector graphics. It was executed in the Windows XP for the comparison with the Hybrid RI. The test was progressed in the intel Core Duo 1.83GHz CPU, 2GB RAM, 2 MB (L2) Cash Memory, ATI Radeon X1400 Graphic card, and 128MB Video Ram environment.



(a) Basic path performance (b) Complex path performance

Fig. 7. Performance test result

The Fig. 7 shows the performance measurement result of Path drawing. The Fig. 7(a) shows the measurement result of drawing basic diagram such as, a triangle or a square. The Fig. 7(b) shows the measurement result of drawing little more complicated figures like an ellipse or a rounded square. As shown in the figure, the OpenVG RI proposed in this paper is faster than the Hybrid RI 1.3-1.6 times in case of the basic path, 4.6-76 times in case of complicated path.

As to the tiger image, the Hybrid RI took 7.3 seconds and our RI did 3.6 seconds, when the size of the drawing surface was 1,024 x 768.

6 Conclusion

The Flash™ already occupies the market more than 80%, in the vector graphics field. The OpenVG was initiated latter than the Flash, but has been settled as the industry standard. We expect that the OpenVG will expand market occupancy sooner or later, because most of the graphic card companies participate in standardization.

In this paper, we proposed the reference implementation of the OpenVG using software rendering. We showed the possibility of success of the software rendering mode, by showing the proposed RI outperforms the Hybrid RI. Moreover, we could port the proposed OpenVG RI easily to the various platforms due to systematically designing it considering an embedded environment.

The academic research or development case of the OpenVG has been scarcely reported because it is introduced recently. We expect that this paper will become the turning point that it activates the OpenVG research and development.

In the future work, we will enhance the success rate of the CTS and continue to research about the performance improvement by using software rendering mode. Moreover, we have a plan to research of supporting the SVG based on the OpenVG.

References

1. Kari Pulli: New APIs for Mobile Graphics, Proceedings of SPIE - The International Soci ety for Optical Engineering (2006), Vol. 6074, art. no. 607401
2. G. He, Z. Pan, C. Quarre, M. Zhang, H. Xu: Multi-stroke freehand text entry method using OpenVG and its application on mobile devices, LNCS (2006) Vol. 3942 791-796
3. Khronos Group Std. OpenVG, Kronos Grouop Standard for Vector Graphics Accelerations, http://www.khronos.org/openvg/, 2005
4. R. Huang and S.-I. Chae: Designing an OpenVG accelerator: algorithms and guidelines, Proc. Int'l Conf. Computer & Communication Engineering (May 2006) 555–560
5. Khronos Group Std. EGL, Kronos Grouop Standard for Native Platform Graphics Interfaces, http://www.khronos.org, 2005
6. Alan Watt: 3D Computer Graphics 3rd Edition, Addison-Wesley (2000)
7. Hybrid Graphics Forum, OpenVG Reference Implementation, http://forum.hybrid.fi (2005)
8. R.C. Gonzalez, R.E. Woods: Digital Image Processing 2nd Edition, Addison-Wesley (1992)
9. Ren Huang, Soo-Ik 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 (2006) 179-184
10. A.Schilling: A new simple and efficient antialiasing with subpixel masks. ACM SIGGRAPH Computer Graphics (July 1991) Vol. 25 No. 4 133–141
11. P.Haeberli and K.Akeley: The accumulation buffer: hardware support for high-quality rendering, ACM SIGGRAPH Computer Graphics (August 1990) Vol. 24 No. 4 309–318
12. K. Doan: Antialiased rendering of self-intersecting polygons using polygon decomposition, Proc. 12th Pacific Conf. Computer Graphics and Applications (2004) 383–391
13. Steven Harrington: Computer Graphics A Programming Approach 2nd Edition, McGraw Hill (2006)
14. KWISF, Wireless Internet Platform for Interoperability, www.wipi.org.kr (2006)
15. Huone, Confermance Test Suite for OpenVG, www.khronos.org (2006)