ISSN 2278-3091
Volume 9 No.2, March -April 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse98922020.pdf
https://doi.org/10.30534/ijatcse/2020/98922020

A Simplified Implementation of the Fixed-Function Graphics Pipeline: DRM Approach

Nakhoon Baek
School of Computer Science and Engineering, Kyungpook National University,
Republic of Korea, oceancru@gmail.com

ABSTRACT

In modern computer graphics architecture, the graphics processing unit (GPU) has a major role. Since GPU instructions and memory managements are hard to be directly controlled by application programmers, three-dimensional graphics application libraries, including OpenGL and DirectX, are widely used. In the Linux-family operating systems, they provide a set of kernel-level supporting modules. In this paper, we use those modules, such as the direct rendering manager (DRM) module, the kernel mode setting (KMS) module, and the graphics execution manager (GEM) module, to implement a fixed-function graphics pipeline. Our prototype system shows that those DRM modules can provide the full 3D graphics features without 3D graphics libraries and graphics window systems. Our detailed design and implementation schemes are presented.

Key words : 3D graphics output, direct rendering manager, fixed-function graphics pipeline, kernel support

1. INTRODUCTION

In these days, there are so many three-dimensional graphics application programs and user interfaces. Since the graphics output devices are widely available, they developed large amounts of graphics applications. Their areas covered various fields including computer animations, computer games, user experiences, human-computer interfaces, and so on [1]–[3].

Historically, we have many kinds of graphics libraries, including OpenGL [4]–[7], X window systems [8], OpenInventor [9], Display PostScript [10], DirectX [11], Cairo [12], Qt [13], and others. Nowadays, we use these 3D graphics libraries very widely for both graphics applications and non-graphics applications.

To produce desired two-dimensional (2D) and/or three-dimensional (3D) images on the screen, they necessarily need the graphics system, and the graphics API (application programming interface) libraries [14], [15]. In general, 3D graphics API libraries and their derived 3D graphics engines are so widely used for developing 3D graphics applications [16]–[18]. Currently, they use OpenGL [4], [5] or DirectX [11] for the underlying 3D graphics API for the graphics engines, and also 3D graphics applications.

Since the graphics libraries developed in a step-wise manner, the current graphics architecture is a kind of library stacks, as shown in Figure 1. In the case of modern OpenGL library implementation, it works over the operating system kernels, X window system, and GLX extensions. Even the two-dimensional graphics output support with the X window system simultaneously work on the same system [6]–[8].

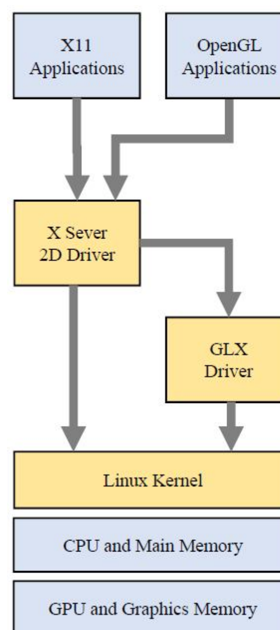

Figure 1: A typical graphics stack on Linux systems.

In this paper, we start from analyzing graphics supports in the Linux kernel, and accelerate the 3D rendering with these features. In this case, we have no need to integrate the

* This work has supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (Grand No.NRF-2019R1I1A3A01061310).

accelerated 3D rendering features with graphics window systems and/or graphics acceleration extensions. We represent a prototype implementation of our fixed-function graphics pipeline, in the following sections.

2. DESIGN STRATEGY

The start point of our idea is that one of the heaviest overheads for the modern windowing systems is the graphical window handling [8], [19] – [22]. It includes the user interfaces and interactions, the event handling, and more additional technical aspects of the overall user interactions. In contrast, some computer graphics architectures adopt direct managing systems, which accesses the frame-buffer directly, as shown in Figure 2.

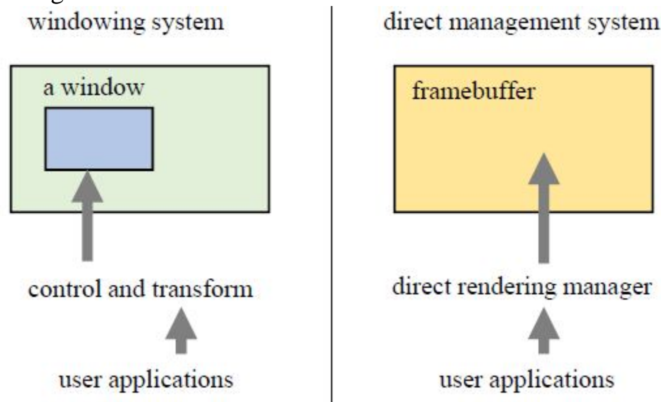

Figure 2: Window systems and direct management of frame-buffers.

2.1 Theoretical Background

From some historical reasons and also some technical reasons, most of the modern 3D graphics library implementations, including OpenGL [4], [5] or DirectX [11], use the normalized device coordinate (NDC) system for its internal reference coordinate system. In this configuration, the NDC coordinates of (x_d, y_d, z_d) are distributed in the interior of the 3D unit cube of $[-1, +1] \times [-1, +1] \times [-1, +1]$, in its NDC system [1], [4], [21].

Assuming that we have an on-screen target viewport, the final device coordinate (x_w, y_w, z_w) of that normalized device coordinate (x_d, y_d, z_d) as follows:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ \frac{f-n}{2} z_d + \frac{f+n}{2} \end{pmatrix}, \tag{1}$$

where (o_x, o_y) is the center point of the viewport, and p_x and p_y are the width and the height of the viewport, respectively. Two additional programmer-specifiable values of n and f, which are applied to the depth value z_d, are the near and the far depth value in the device coordinate system [6], [21].

With the device coordinates, we still have some technical issues on the rasterization of the graphics primitives. For more advanced applications including anti-aliasing and texture mapping, we precisely calculate the per-fragment coordinate of (x_p, y_p), where x_p and y_p varies from 0 to 1 in the x-direction and y-direction, respectively. The per-fragment coordinate value (x_p, y_p) are calculated as follows:

$$\begin{pmatrix} x_p \\ y_p \end{pmatrix} = \begin{pmatrix} \frac{1}{2} + \frac{x_f + \frac{1}{2} - x_w}{\text{size}_x} \\ \frac{1}{2} + \frac{y_f + \frac{1}{2} - y_w}{\text{size}_y} \end{pmatrix}, \tag{2}$$

where size_x and size_y are the fragment size, horizontally and vertically, respectively. The parameters x_f and y_f are the integer coordinate values of the given fragment. The device coordinate value (x_w, y_w) is the unrounded window coordinate of the input vertex.

For more precise calculations, including texture mapping and level-of-detail operations, they need to calculate the detail levels of graphics primitives. As an example, in the case of edges, the edge length s can be calculated as:

$$s = \frac{r}{f} \cos^{-1}\left(\frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{|\mathbf{v}_1||\mathbf{v}_2|} \right), \tag{3}$$

where the resolution r is specified in pixels, and f is the field of view. The vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are measured from the camera positon to the vertices of the given edge. After its calculation, we can choose the suitable level-of-detail factors, with respect to the edge length s [24].

2.2 Kernel-level Graphics Support

For the Linux-family operating systems, kernel-level modules are developed to support modern graphics features, along to its development. Although they call those set of modules as device drivers, the modern 3D graphics drivers are complicated and large-size, in comparison with other device drivers. The core modules of direct rendering manager (DRM) [25], [26], kernel mode setting (KMS) [27], and graphics execution manager (GEM) [28] will be explained.

Historically, the direct rendering manager (DRM) module was introduced to access the frame-buffers directly. With the legacy graphics architectures, this direct access capability is enough to support the simple graphics features, such as 2D graphics and graphics window handling. Later, they need to support 3D graphics pipeline, and finally, they found that those 3D graphics features need isolated processors, in addition to the central processing unit (CPU).

Thus, in modern computer graphics architecture, they introduce the new processing unit, named the graphics-processing unit (GPU). The modern GPU is actually a specialized parallel-processing unit, with independent graphics memory and the frame-buffer memory. The DRM module now manages the whole set of GPU, its graphics memory, and the whole frame-buffer [25].

In the Linux programming, the DRM is realized as an application program interface (API) [26]. The upper layer API's, which typically support the high-level 3D graphics API functions, use this low-level API, to transfer the GPU instructions and graphics data to the GPU. Figure 3 shows the typical uses of the DRM in Linux systems, especially with the OpenGL graphics library.

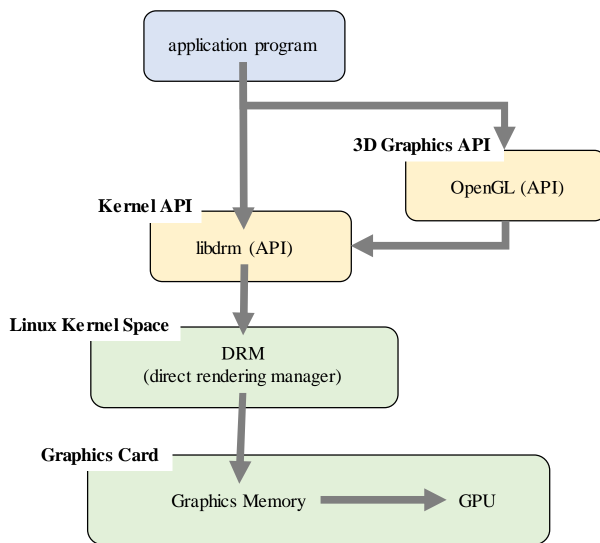

Figure 3: The DRM module in the Linux kernel.

In these days, the widely used GPUs typically have a few mode setting command to allow the selection of their important internal features. Typical examples include the screen resolution changing, depth-bit configuration, stencil-bit configuration, screen refresh-rate changing, color-buffer configuration, and more technical settings. For a 3D graphics applications, they will send these mode setting commands first, and then, typical 3D drawing commands, sequentially to the GPU.

As already mentioned, the Linux and its derived systems provide their kernel features typically with a set of specific-purpose modules. Among them, the kernel mode setting module (or shortly, KMS) is designed to provide the GPU mode setting commands [19] – [24]. For 3D graphics application programs, they can use these KMS features to manage the GPU and screen configurations directly, or can use other 3D libraries and 3D rendering engines to indirectly use these features.

From the GPU point of view, the graphics memory is one of the most important resources, and they should be controlled carefully. Even for a single GPU, it should support a set of 3D graphics applications simultaneously. It means that the GPU should control its own graphics memory against several 3D graphics applications. In modern graphics applications, the term of the graphics context is widely used to refer the current 3D graphics settings, and the graphics memory handling information for a specific 3D graphics application [27].

To support the graphics memory handling with respect to the graphics contexts of the 3D graphics applications, the Linux family operating systems also have the graphics execution manager module (shortly, GEM), in their kernel configurations. The GEM module also has the features to share the graphics memory areas between different 3D graphics programs. With the modern GPU features, the GEM module is emphasized to focus on the graphics context management, in addition to its original low-level graphics memory controls [28].

In modern 3D graphics applications, they consume more and more graphics memory, for their texture images, modeling features, and other graphics-related information. Those graphics data, including texture images and also photo-scanned images, should be delivered to the GPU-controlled graphics memory, from the hard disk or main memory area, for each start time of every graphics context. To improve the 3D graphics application performances, those graphics data has better to be persistently managed, even for the context switching, if possible. This persistent data handling will improve the performance of graphics applications.

As other modern operating systems, the Linux-family operating systems also provide the plentiful 3D rendering features, to be used as the basic, and additionally, the advanced tools for graphics and image-processing related applications. Typical 2D and 3D graphics and image-processing applications process their data into the off-screen buffers (or back-buffers), and the final image will be mapped onto the screen (or front-buffers). As previous mentioned, the KMS module and GEM module provides all the managing operations and graphics data handling operations for these applications, in the case of Linux-family operating systems [28].

Based on these Linux kernel-level modules, we found the possibility that a 3D graphics application program can avoid the use of graphics window systems such as the X window system, and also avoid the 3D graphics application-level libraries, such as OpenGL. We have designed the overall layout for our 3D direct rendering system, based on the DRM approach [19]. In this paper, we also present the implementation result of our prototype system, in the following sections.

3. IMPLEMENTATON RESULTS

Based on our intuitive idea, we implemented a prototype graphics rendering system. For simplicity, we focused on the rendering pipeline itself, rather than any programmable features. Therefore, we selected the OpenGL fixed function graphics pipelines, as shown in Figure 4.

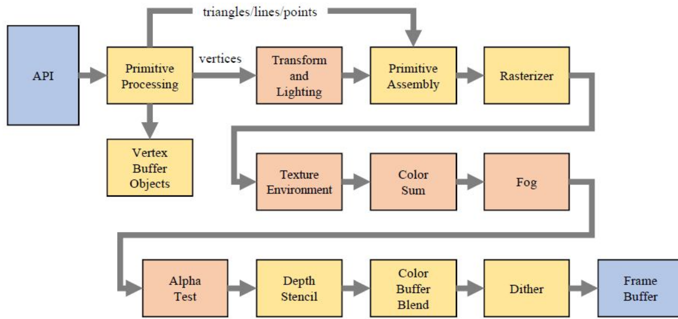

Figure 4: An example fixed-function graphics pipeline of the OpenGL family.

Our system has the kernel of 3D graphics-transformations and rendering features. To derive modern GPU core instructions, we use the fixed GPU instruction sequences extracted from the existing implementations of OpenGL [4], [6].

Figures 5 and 6 shows the results of our DRM-based direct 3D graphics rendering system. As shown here, existing 3D graphics-output routines successfully works with our system. Animation sequences are also successfully working with our systems. Texture mapping and alpha blending also work well.

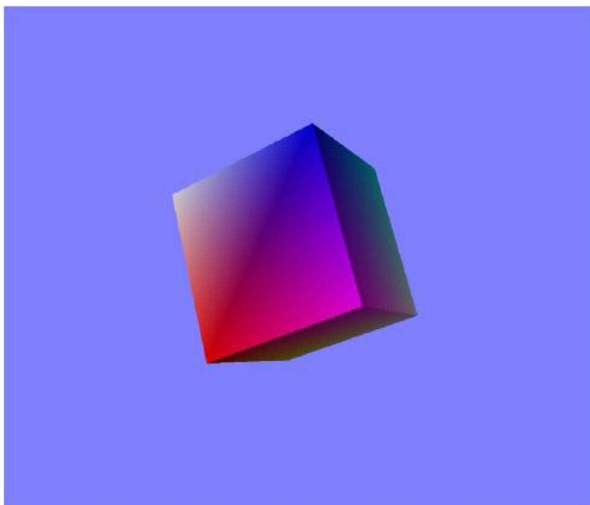

Figure 5: An example output from our DRM-based 3D graphics system.

Figure 6: Another example from our DRM-based 3D graphics system.

4. CONCLUSION

In these days, almost all 3D graphics applications use the commercial implementation of 3D graphics libraries and/or 3D graphics engines. Due to historical reasons and other factors, those commercial 3D graphics implementations typically works on the full-scale graphics stacks, which includes the graphics window system and other managing features.

In this paper, we represent a simplified 3D rendering system, which directly uses some Linux operating system kernel modules, including KMS and DRM modules. Since the Linux DRM module provides the low-level graphics features as the kernel services of the Linux operating system, it was difficult to get the full features of high-level graphics application requirements. As the first step, we focused on the simple and intuitive fixed-function graphics operations. Our prototype system finally provides the whole features of simplified fixed function 3D graphics rendering functions.

We expect that our current implementation is a possible solution for time-critical big-size 3D data visualization applications, especially in the big-data visualization area. Extending this system toward standardized implementations, such as OpenGL ES 1.1 or OpenGL SC 1.1, may be our next steps.

REFERENCES

1. J. F. Hughes, et al., ***Computer Graphics: Principles and Practice***, 3rd Edition, Addison-Wesley, 2013.
2. N. Baek and K. Ryu, **Emulating OpenGL ES 2.0 over the desktop OpenGL**, *Cluster Computing*, Vol. 18, pp. 165–175, 2015. https://doi.org/10.1007/s10586-014-0351-6
3. N. Baek, **An emulation scheme for OpenGL SC 2.0 over OpenGL**, *The Journal of Supercomputing*, (online-first), 2020.

4. M. Segal and K. Akeley, ***The OpenGL Graphics System: A Specification***, Version 4.5 (Core Profile), Khronos Group, 2016.
5. J. Kessenich, ***The OpenGL Shading Language***, Language Version: 4.50, Khronos Group, 2016.
6. A. Munshi and J. Leech, ***OpenGL ES Common Profile Specification***, Version 2.0.25 (Full Specification), 2010.
7. R. J. Simpson, ***The OpenGL ES Shading Language***, Language Version: 1.00, 2013.
8. D. Young, ***The X Window System: Programming and Applications with Xt, OSF/Motif***, 2nd Edition, Prentice Hall, 1994.
9. J. Wernecke, ***The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor***, Addison-Wesley, 1994.
10. Adobe Systems, ***Programming the Display Postscript System With X***, Addison-Wesley, 1993.
11. F. Luna, ***Introduction to 3D Game Programming with DirectX 12***, Mercury Learning & Information, 2016.
12. cario website, http://www.cairographics.org/, 2020.
13. G. Lazar, ***Mastering Qt 5***, Packt Publishing, 2017.
14. A. Malizia, ***Mobile 3D Graphics***, Springer-Verlag New York, 2006.
15. K. Pulli, et al., ***Mobile 3D Graphics: with OpenGL ES and M3G***, Morgan Kaufmann Publishers Inc., 2007.
16. M. Kumar, S. Inthiyaz, J. Dhamini, A. Sanjay and U. Srinivas, **Delay Estimation of Different Approximate Adders using Mentor Graphics**, *International Journal of Advanced Trends in Computer Science and Engineering*, Vol. 8, pp. 3584–3587, 2019. https://doi.org/10.30534/ijatcse/2019/141862019
17. A. Africa, C. Alcantara, M. Lagula, A. Latina, Jr., and C. Te, **Mobile Phone Graphical User Interface (GUI) for Appliance Remote Control: An SMS-based Electronic Appliance Monitoring and Control System**, *International Journal of Advanced Trends in Computer Science and Engineering*, Vol. 8, pp. 487–494, 2019. https://doi.org/10.30534/ijatcse/2019/23832019
18. N. Hussin and W. Li, **A modified Particle Swarm Optimization algorithm linking dynamic neighborhood topology to parallel computation**, *International Journal of Advanced Trends in Computer Science and Engineering*, Vol. 8, pp. 112–118, 2019. https://doi.org/10.30534/ijatcse/2019/03822019
19. N. Baek, **A Fixed-Function Rendering Pipeline with Direct Rendering Manager Support**, in *Proc. ICITCS 2017*, pp. 106-109, 2017.
20. N. Baek and K. Kim, **Design and Implementation of OpenGL SC 2.0 Rendering Pipeline**, *Cluster Computing*, Vol. 22, sup.1, pp. 931–936, 2019. https://doi.org/10.1007/s10586-017-1111-1
21. K. Kim and N. Baek, **Providing Profiling Information for OpenGL ES Application Programs**, *Cluster Computing*, Vol. 22, sup.1, pp. 937–941, 2019.
22. N. Baek and K. Kim, **Prototype Implementation of the OpenGL ES 2.0 Shading Language Off-line Compiler**, *Cluster Computing*, Vol. 22, sup.1, pp. 943–948, 2019.
23. H. Guo, ***Modern Mathematics and Applications in Computer Graphics and Vision***, World Scientific Pub, 2014. https://doi.org/10.1142/8703
24. J. Mueller et al., **Shading Atlas Streaming**, in *Proc. ACM Siggraph Asia 2018*, 2018.
25. R. E. Faith, **The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure**, http://dri.sourceforge.net/doc/drm_low_level.html, 2020.
26. J. Fonseca, **Direct Rendering Infrastructure: Architecture**, https://paginas.fe.up.pt/~mei04010/dri-architecture.pdf, 2005.
27. **kernel mode setting**, https://wiki.archlinux.org/index.php/kernel mode setting, 2018.
28. K. Packard and E. Anholt, **The Graphics Execution Manager: Part of the Direct Rendering Manager**, https://lwn.net/Articles/283798/, 2008.