A Prototype Implementation of a CUDA-based Customized Rasterizer

Nakhoon Baek
School of Computer Science and Engineering
Kyungpook National University
Daehak-ro 80, Daegu 41566, Korea

***Abstract*—In these days, we have high-performance massively parallel computing devices, as well as high-performance 3D graphics rendering devices. In this paper, we show a prototype implementation of a full-software 3D rasterizer system, based on the CUDA parallel architecture. While most of previous CUDA-based software rasterizer implementations focused on the triangle primitives, our system includes more 3D primitives, and extra 2D primitives, to fully support 3D graphics library features. Currently, our system is at its prototype implementation stage, and it shows successful results with 3D primitive handling and also character output features. Our design and implementation details are presented. More optimizations and fine tunes will be followed in near future.**

Keywords*—3D rasterization; CUDA implementation; OpenGL emulation

I. Introduction

Web3D applications are designed to fully display and navigate web sites using 3D graphics features. Fundamentally, the Web3D applications need 3D rasterization process, either on the web-browser side, and/or on the web-server side. In this paper, we present a prototype implementation of 3D rasterizer, based on massively parallel processing features, as a framework for 3D web and associated application domains.

Massively parallel computing features are now widely available in many areas of computer science and engineering. From the 3D graphics rendering point of view, the 3D rendering pipelines are naturally developed to use massively parallel processing features. Additionally, we can also use the massively parallel computing features, especially with CUDA (compute unified device architecture) [1] and OpenCL (open computing language) [2].

Since these massively parallel pipelines, the 3D graphics pipeline and the high-performance computing pipeline, have many common characteristics, there have been several works to integrate these two different pipelines into a single one. More precisely, they tried to implement the 3D graphics rendering pipeline, on the existing parallel computing pipeline [3].

In previous works, they focused on the feasibility test, and most of them provides mainly the triangle rasterization process, with massively parallel computing libraries. In contrast, we aim at the full-scale 3D graphics rendering library implementation. For example, to provide the full features of the OpenGL (open graphics library) system, we need much more rather than the triangle rasterizer. At this time, we have a prototype implementation, which shows the possibility of the CUDA-based rasterizer, with several 3D graphics primitives and also extra 2D graphics primitives. It is the distinguished point of our work, in comparison to the previous works.

We start from presenting the previous works in Section II. Our motivation and overall design of the 3D rendering system based on CUDA will be presented in Section III. Implementation details and results from the prototype implementation will be followed in Section IV.

II. Previous Works

In 1990's, the programmable graphics pipeline has been introduced [4]. Rapidly, the GPU (graphics processing unit) became a computing device, with the concepts of GPGPU (general purpose GPU). In 2000's, the massively parallel computing devices including CUDA [1] and OpenCL [2] are available.

Since the programmable graphics pipeline and the massively parallel computing pipeline have common features, there have been several trials to implement the graphics processing features on the massively parallel processing devices. Some of them are summarized in the followings.

Mesa 3D graphics library [5] was originally implemented with CPU computing powers. Mesa is actually started as a re-implementation of widely-used 3D graphics libraries, including OpenGL [6] and Vulkan [7]. In its components, Mesa provides a full-software implementation of rasterizers, called "swrast". This implementation enables the 3D graphics rendering and 3D graphics shader features on CPUs. However, this CPU-based implementation is very slow, as we can expect, and thus, used only for limited purposes. Since this implementation was already available in 1990's, it actually affected its following implementations of software 3D graphics rasterizers.

Intel Larrabee project [8] was actually a hardware architecture, while it also aimed to provide an efficient software implementation of the 3D graphics rendering features. From the parallel processing point of view, the Larrabee project implements a binned renderer to increase the parallel processing features, and to reduce the memory bandwidth. Unfortunately, the Larrabee project was cancelled, and later rearranged to make high-performance computing processors.

FreePipe [9] is a fully-programmable 3D graphics pipeline, implemented in CUDA programming library. It developed some special features for the efficient rendering, even in a single pass rendering, with CUDA atomic operations. It shows good performance for small-size objects, while the

performance drops rapidly for large-scale and/or large-size objects, mainly due to the CUDA atomic operation behaviors.

CUDARaster [10] and its followers [11], [12], [13] are also software 3D graphics rendering pipeline implementations, with CUDA. CUDARaster was implemented for a specific CUDA model of Fermi, and it also uses some assembly-level codes, for optimization purpose. Unfortunately, it cannot be executed on the new CUDA architectures, since it was highly tuned and dependent on the old CUDA architecture.

The cuRE [14] is another rasterizer implementation to resolve the drawbacks of the previous implementations. This new rasterizer architecture can be executed on various modern CUDA architectures. It also shows several modifications, including direct wireframe rendering, programmable blending, and others.

Although we have some rasterizer implementations, especially based on the CUDA parallel processing architecture, our work aims to finally implement the full-scale 3D graphics system. Thus, we will include more 3D graphics primitives, and also 2D graphics primitives. The design and implementation of our system will be presented in the following sections.

III. Overall Design

In the case of OpenGL, they need at least the following 3D primitives:

- Points: A set of 3D points can be displayed. Additionally, they can set the radius of the point, or equivalently, the point size. The initial point size of 1 means a single pixel point, while we can also specify more big size points, which will be displayed as circles or rectangles on the screen.
- Line segments: A pair of 3D points can specify a 3D line segment. A sequence of line segments can also be displayed. In most cases, they use the line width of 1, to show one pixel wide line segments. With larger line widths, we can display thick line segments.
- Triangles: A set of three 3D points can define a triangle. A sequence of triangles are also possible, with triangle strips and triangle fans. Those triangles are usually filled with specified colors, or texture images.

In the previous works, they concentrated on the triangle primitives. In fact, the CUDA-based implementation of the triangle primitive is sufficiently difficult work, to be optimized and finely tuned. Also, we need to consider that most of 3D graphics scenes are constructed with triangles, since modern computer graphics object models are mostly based on the 3D triangle mesh models.

For a full-scale 3D graphics library implementation, we naturally need all of these 3D output primitives: points, line segments, and triangles. Additionally, for practical reasons, some 2D primitives are also needed to full-scale implementations. As an example, the resetting or updating of 2D rectangular areas in the framebuffer areas and/or in the texture image areas are needed frequently, even for the 3D graphics libraries.

Mostly required 2D operations are actually *bit-blt* (bit block transfer) operations, and can be summarized as follows:

- rectangular fill: The given rectangular 2D framebuffer (or image) area will be updated with the given colors (or numerical values). This operation can be used for the clearing of the whole or any partial framebuffer area.
- pixmap bit-blt: A pixmap means a 2D array layout of pixel values. A colorful image can be the typical cases. The image will be transferred (or more precisely, copied) to the specified rectangular area in the framebuffer or in the texture image area.
- grayscale bit-blt: A grayscale image can also be transferred to the framebuffer area.
- bitmap bit-blt: A bitmap represents a black/white image, through representing a black/white pixel with a single bit. This format is frequently used for bitmap fonts. Through transferring the bitmaps on the screen, we can display characters on the screen for extra information display.

For the overall design of the system, the 3D output pipeline will be maintained as the main stream pipeline. Fig. 1 shows the full 3D graphics pipeline of OpenGL 2.0 and OpenGL ES 2.0, which supports the vertex shaders and fragment shaders. It is used as the start point of our implementation. Our current prototype implementation focuses on the primitive assembly and rasterizer module.

The essential role of the "primitive assembly and rasterizer" module is converting the given 3D coordinate specifications to a set of 2D pixels, which are targets to be updated. The vertex shader and the fragment shader can be regarded as the preprocessing and post-processing to this module.

At this time, our CUDA-based implementation is based on the commercial CUDA-capable graphics cards. Thus, our prototype implementation is realized as a set of CUDA kernel programs, as shown in Fig. 2.

User inputs and rendering operations are provided through the C/C++ API function calls, to prepare the 3D graphics data in the CPU memory area. The 3D graphics data will be copied to the CUDA memory area, similar to the typical CUDA programs.

A big-size CUDA memory area is dedicated to the "logical framebuffer", which act as the real framebuffer, but cannot be displayed directly on the screen. Instead, the "logical framebuffer" is shared as an OpenGL texture image, and an independent OpenGL program is executed to simply display the logical framebuffer texture image on the screen.

For embedded systems, we can customize the current implementation, with new hardware display circuit supports, as shown in Fig. 3. With customized display logic implementations, the "logical framebuffer" can act as the real physical framebuffer. In this case, the display speed will be much more enhanced. This customized display circuit support will be the future works. In the next section, we will explain our current CUDA kernel implementations.

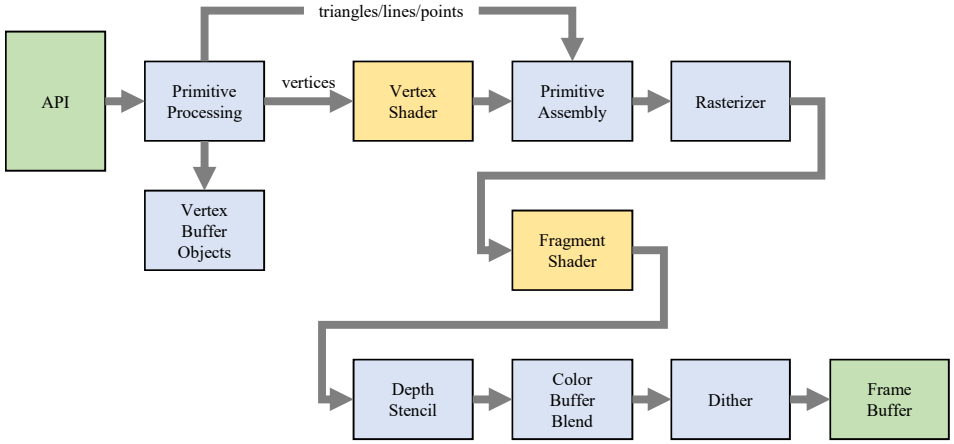


Fig. 1. A Typical 3D Graphics Rendering Pipeline.

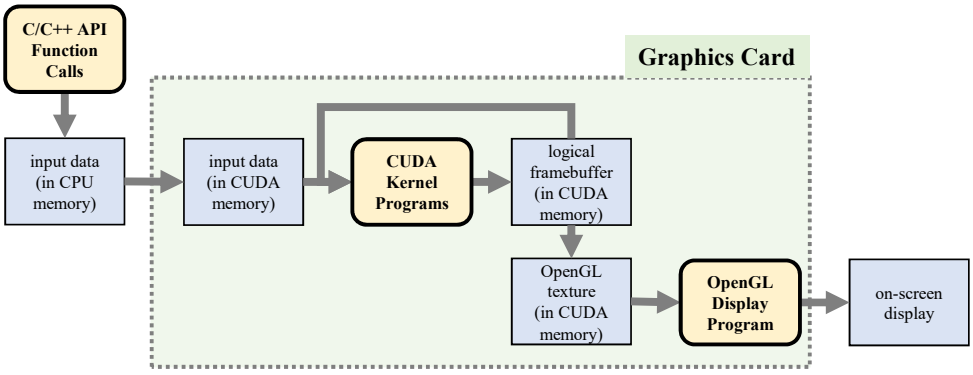


Fig. 2. Our Current CUDA-based Implementation Layout.

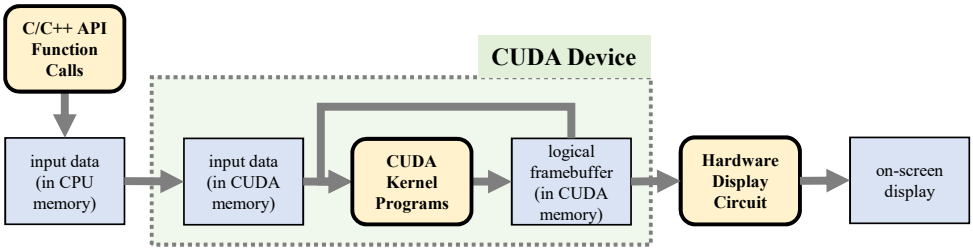


Fig. 3. Another Possible CUDA-based Implementation Layout.

IV. Implementation

The core of our implementation is a set of CUDA kernel programs, whose layouts are based on the rectangular division of the screen. It is actually typical approaches used in most previous works. As shown in Fig. 4, the whole screen (as an example, 1,280 by 1,024 pixels) is divided into a set of rectangular tiles. Each tile consists of 32 by 32 pixels, or equivalent, 1,024 pixels.

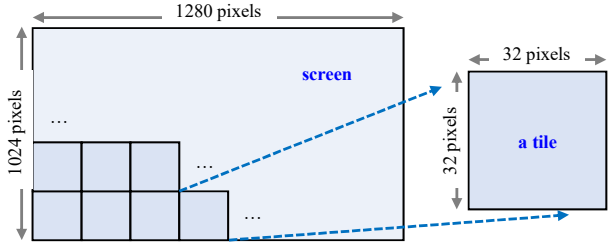


Fig. 4. The Screen Layout with Tiles.

From the CUDA programmer's point of view, it is convenient to allocate a single thread for a pixel on the screen. Thus, as shown in Fig. 5, we use a 2D thread block of 32-by-32 thread layout. This 1,024 threads are actually the current CUDA limits to the maximum number of threads in a single thread block. Then, to make the whole 1,280-by-1,024 threads,

we use 40-by-32 blocks for the CUDA grid. Thus, the whole grid corresponds to the whole screen. The grid is divided into 40-by-32 thread blocks, while the screen divided into that numbers of tiles. And, the 32-by-32 thread block corresponds to the thread block.

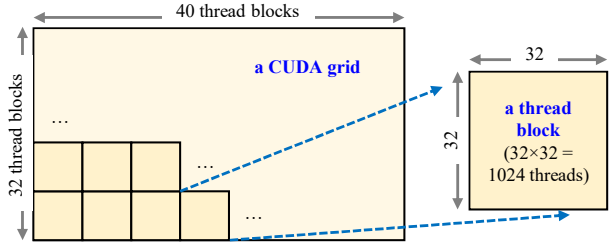
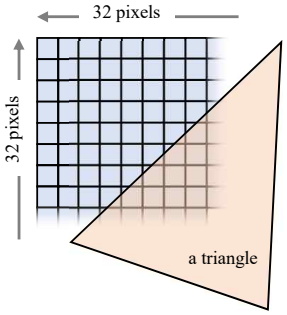
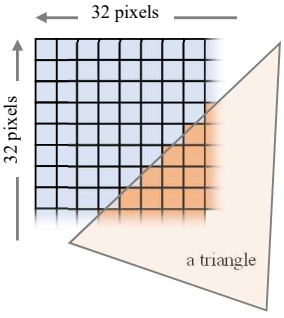


Fig. 5. The CUDA Grid Layout with Thread Blocks.

This thread block layout has some benefits. In the case of triangles, the pixels, or equivalently, the CUDA threads can decide whether they are located in the interior of a given triangle or not, in a massively parallel way. Each thread will calculate the signed areas of some configurations, from the given window coordinates of the vertices. Only the interior threads will turn on their corresponding pixels, to display the given triangle on the screen, as shown in Fig. 6.



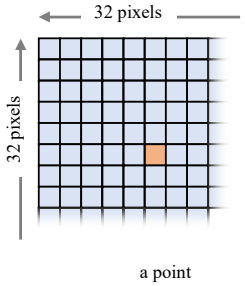
(a) A Triangle Overlaid on the Tile



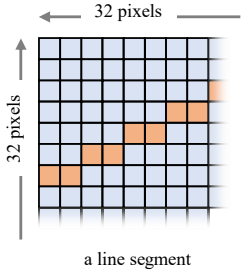
(b) The Triangle Rasterization Result

Fig. 6. A Tile-Based Rasterization Example for a Triangle.

In the case of points and line segments, the tile based approach can be inefficient, especially for the single pixel points and the single pixel wide line segments. For a single pixel point, the thread block should launch totally 1,024 threads, due to the CUDA kernel launch mechanism and our thread block configurations. Then, only one thread will turn on the pixel, while others all should discard their processing, as shown in Fig. 7(a). Similarly, a single pixel wide line segments, at most 32 pixels will be turned on, even though initiating totally 1,024 threads, as shown in Fig. 7(b).



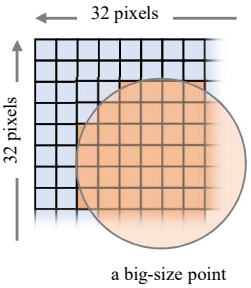
(a) A Single-Pixel Point Case



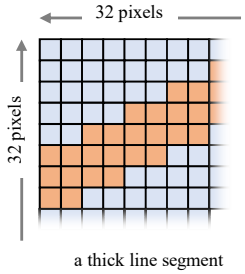
(b) A Line Segment with One Pixel Width

Fig. 7. A Tile-Based Rasterization Example for a Point and a Line Segment.

In contrast, the tile based approach can efficiently works with big-size points, and think line segments. As shown in Fig. 8(a), the big-size points are typically implemented as circles, and the threads in the tile can check whether they belongs to the interior of the circle or not, similar to the triangle cases. Thick line segments can also be handled efficiently, as shown in Fig. 8(b). The effective threads can check their corresponding conditions in a massively parallel manner.



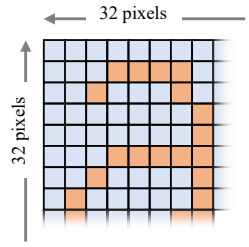
(a) A Big-Size Point Case



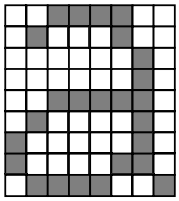
(b) A Thick Line Segment Case

Fig. 8. Another Tile-Based Rasterization Example.

The tile-based approach can work for most of bit-blt operations. For a given rectangular region, the threads will get the corresponding pixel information, and then update their own pixels, to get the final result. As a direct application of these bit-blt operations, we added character display features to our implementation. In this case, the true type fonts are pre-processed to get the character font information and the grayscale image of each character [15], as shown in Fig. 9(b). Our thread blocks will process the character font information, and finally show the character on the screen, as shown in Fig. 9(a).



(a) A Character Display Example



(b) A Grayscale Image from True Type Fonts

Fig. 9. A Tile-Based Rasterization for Character Display.

At this time, our prototype implementation shows the basic rasterization CUDA kernels are working well. As an example, Fig. 10 shows the screen shot of the triangle rasterization result, from our CUDA-based rasterizer implementation. It shows the correct display of the triangle coordinates, as specified in the input vertex specifications.

Additionally, the barycentric interpolation of interior points are also demonstrated. We specified different vertex colors at each vertex of the triangles. The interior pixels have the interpolated colors, according to the barycentric interpolation, specified in the OpenGL specification [6].

Unlike the previous works, our CUDA-based rasterizer implementation supports more output primitives, in addition to the triangle primitives. Fig. 11 shows a demonstration of 3D points, from our prototype implementation. It shows the circular points, as expected.

As another distinct example, we implemented the text output routines, with underlying bit-blt primitive support. Our CUDA kernels support bit-blt operations, and we use some grayscale or bitmap images of the true type fonts, with the free true type font library [15]. The images are blended into the screen, to make smooth font display results. Fig. 12 shows an example screen shot of our font rendering result, with more than 100 text output results, each of which specify random text colors and a complete sentence to be displayed. It shows that our CUDA-based implementation has some

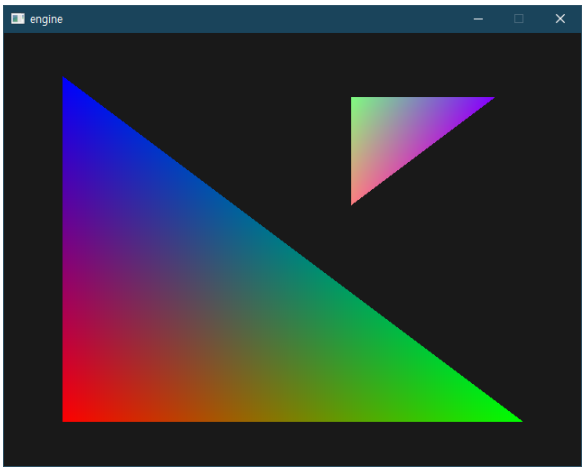


Fig. 10. A Screen Shot of Triangle Rasterization, from our CUDA-Based Rasterizer Implementation.

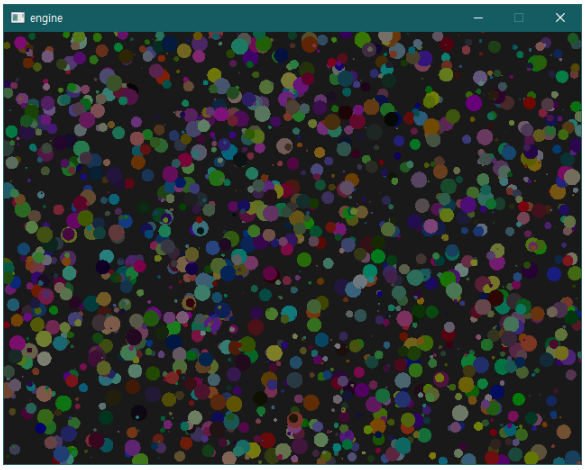


Fig. 11. A Screen Shot of Point Rasterization, from our CUDA-Based Rasterizer Implementation.

distinguished points, in comparison to the previous rasterizer implementations.

V. Conclusion

Our motivation was implementing the 3D graphics rendering pipeline on the massively parallel computing pipeline. In this case, we can make a full-software implementation of the graphics rendering features. To realize this goal, we started to implement common 3D rendering features on the CUDA architecture.

Since we aimed to get a full-scale implementation of typical 3D graphics library, we selected several 3D graphics primitives including points, line segments and triangles. Additional image handling operations are also needed, and we added some 2D pixel level primitives. Currently, our prototype implementation shows those primitives are working well. More fine tunings and optimizations should be followed, and they will be our near future works.

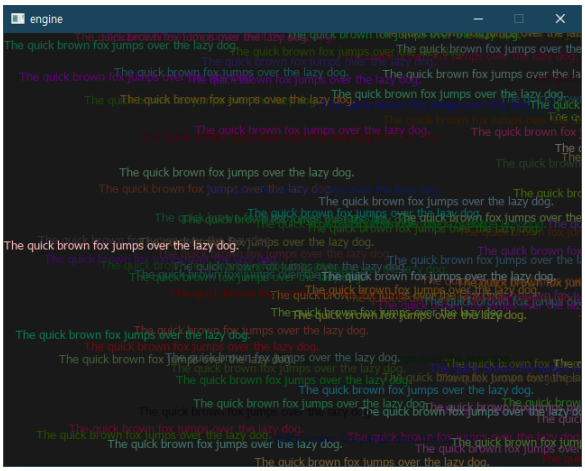


Fig. 12. A Screen Shot of True-Type Font Rasterization, from our CUDA-Based Rasterizer Implementation.

ACKNOWLEDGMENT

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).

This study was supported by the BK21 FOUR project (AI-driven Convergence Software Education Research Program) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (4199990214394).

REFERENCES

[1] NVIDIA, *CUDA Toolkit Documentation, version 11.7.0*. NVIDIA, 2022.

[2] Khronos OpenCL Working Group, *The OpenCL Specification, version 3.0*. Khronos Group, 2022.

[3] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogene-ous computing systems," *Computing in science & engineering*, vol. 12, no. 66, 2010.

[4] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proc. of the 6th Int'l Symp on Memory Management (ISMM '07)*. ACM, 2007, pp. 103–104.

[5] Mesa Team, *The Mesa 3D Graphics Library*, retrieved in July 2022. [Online]. Available: http://www.mesa3d.org/

[6] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification, version 4.6*. Khronos Group, 2019.

[7] The Khronos Vulkan working group, *Vulkan - A Specification, version 1.3.223*. Khronos Group, 2022.

[8] L. Seiler *et al.*, "Larrabee: A many-core x86 architecture for visual computing," *IEEE Micro*, vol. 29, pp. 10–21, 2009.

[9] F. Liu, M. C. Huang, X. H. Liu, and E. H. Wu, "Freepipe: A programmable parallel rendering architecture for efficient multi-fragment effects," in *Proc. of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '10)*. ACM, 2020, pp. 75–82.

[10] S. Laine and T. Karras, "High-performance software rasterization on gpus," in *Proc. of the ACM SIGGRAPH Symp on High Performance Graphics (HPG '11)*. ACM, 2011, pp. 79–88.

[11] Y. C. Kwon and N. Baek, "A cuda-based implementation of opengl-compatible rasterization library prototype," in *Proc. of the 29th Annual ACM Symp on Applied Computing (SAC '14)*. ACM, 2014, pp. 1747–1748.

[12] N. Baek and K. Kim, "Design and implementation of opengl sc 2.0 rendering pipeline," *Cluster Computing*, vol. 22, pp. 931–936, 2019.

[13] M. Kim and N. Baek, "A 3d graphics rendering pipeline implementation based on the OpenCL massively parallel processing," *Journal of Supercomputing*, vol. 77, pp. 7351–7367, 2021.

[14] M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger, "A high-performance software graphics pipeline architecture for the GPU," *ACM Trans. Graph.*, vol. 37, pp. 140:1–140:15, 2018.

[15] FreeType, *The FreeType project*, retrieved in July 2022. [Online]. Available: http://www.freetype.org/