SCIPEDIA

Design and Implementation of a Vulkan-Based Rasterization System in the PC Environment

Mingyu Kim[1] and Nakhoon Baek[1,2,*]

[1] School of Computer Science and Engineering, Kyungpook National University, Daegu, 41566, Republic of Korea

[2] Data-Driven Intelligent Mobility ICT Research Center, Kyungpook National University, Daegu, 41566, Republic of Korea

INFORMATION

Keywords:
Vulkan
local illumination
rasterization
design
implementation

DOI: 10.23967/j.rimni.2025.10.70899

SCIPEDIA Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería

Revista Internacional Métodos numéricos para cálculo y diseño en ingeniería RIMNI In cooperation with CIMNE

Design and Implementation of a Vulkan-Based Rasterization System in the PC Environment

Mingyu Kim[1] **and Nakhoon Baek**[1,2,*]

[1]School of Computer Science and Engineering, Kyungpook National University, Daegu, 41566, Republic of Korea

[2]Data-Driven Intelligent Mobility ICT Research Center, Kyungpook National University, Daegu, 41566, Republic of Korea

ABSTRACT

Graphics application programming interfaces (APIs) for three-dimensional (3D) rendering have been employed in computer graphics. Traditional 3D graphics APIs, such as Open Graphics Library (OpenGL), have structured the entire pipeline for the programmer's convenience and are relatively easy to use. With the advancement of graphics hardware technology, new graphics APIs, including Vulkan, have been introduced to control specific function units of the graphics card and reveal the graphics processing power. Vulkan has many advantages, such as graphics processing power and parallel processing support. However, it also has the disadvantage of increasing the implementation costs because it requires detailed controls. This paper aims to implement a rasterization pipeline that applies a local illumination model with Vulkan. The implemented system can be used for standardized 3D graphics tasks as is and can also be used as a starting point for varying the pipeline configurations. This work designs and implements a Vulkan-based local rasterization pipeline. With this implemented platform, practical 3D rendering scenes are executed to be compared and analyzed with the rendering results from traditional OpenGL. The experimental results demonstrate the correctness and efficiency of the implementation. The implementation will be used as a testbed for various rendering experiments in the future.

OPEN ACCESS

Received: 27/07/2025

Accepted: 11/10/2025

Published: 23/01/2026

DOI
10.23967/j.rimni.2025.10.70899

Keywords:
Vulkan
local illumination
rasterization
design
implementation

1 Introduction

1.1 Drawbacks of Traditional APIs

Three-dimensional (3D) graphics output is widely used for typical 3D applications, virtual reality applications, realistic renderings, and other applications. The 3D graphics rasterization processes have primarily been employed for 3D interactive applications or computer games for more realistic graphics output. This work aims to present a new design and implementation of the 3D graphics rasterization process with one of the latest graphics application programming interfaces (APIs), *Vulkan* [1,2]. In computer graphics, several widely used de facto standard libraries exist, including Open Graphics Library (OpenGL) [3,4], DirectX [5], and Vulkan [1]. Among them, Vulkan is one of the latest and has been designed to replace OpenGL completely.

*Correspondence: Nakhoon Baek (nbaek@knu.ac.kr).

Traditional graphics libraries, such as OpenGL, have some drawbacks. First, the traditional graphics API was initially designed for *fixed-function* graphics processing units (GPUs). To ensure that the existing graphics API operates on the current *programmable* GPUs, GPU vendors have added many pre- and post-processing operations at the device driver level. Thus, performance differences occur for the same application for each GPU vendor, and performance differences occur each time the GPU driver is updated. When a GPU device driver is updated, its performance may increase or decrease. Additionally, shader code processing methods also differ depending on the GPU vendor. In the worst case, an application may run normally on one GPU but not on another GPU [6].

Mobile devices are now widely used, and unlike computer GPUs, mobile GPUs are designed to support *tile-based rendering*. The traditional graphics APIs do not consider mobile GPUs, making the performance worse. Graphics APIs designed for fixed-function GPUs are unsuitable for modern mobile GPUs. Last, the traditional graphics APIs support only *single-thread* execution models. In this case, the CPU computing power may be a bottleneck, making the GPUs work inefficiently.

1.2 Need for a New API, Vulkan

The new graphics API Vulkan was designed to resolve the drawbacks of traditional APIs. Currently, the Vulkan graphics API has the following characteristics compared with other traditional graphics APIs. Vulkan has adopted *verbose programming*, where programmers write code details to control graphics devices. The clear advantage of Vulkan is that application performance may be dramatically increased because unnecessary parts used by the device driver can be removed. However, tremendous effort is required to make even a simple application program. Hence, a triangle cannot be drawn with a few lines of code, as in OpenGL.

Vulkan has many strong points concerning traditional graphics APIs. However, its verbose programming concept makes graphics programming more complicated and is not yet widely used. Although it is widely known that Vulkan can execute a much wider variety of graphics work fast and efficiently, it is currently challenging to find the results of various benchmarks. Vulkan uses a single shader compiler to generate its byte code. All GPU vendors can use the same shader code, and programmers can enjoy increased portability. Vulkan supports *multithread* programming models.

This paper presents a prototype of a *local illumination framework* to test whether existing OpenGL-based programs can be regenerated more efficiently on Vulkan. The graphics framework can be used immediately for various 3D graphics applications and experiments on the graphics pipelines. The following sections present the design and implementation details.

2 Related Work

Although 3D graphics features are widely used, even with the best performance, the current rasterization process has difficulty presenting light effects, such as shadows, reflection, and refraction, in real time. Many researchers have tried to determine a solution using general-purpose computing on GPUs (GPGPU), low-level APIs, and hybrid rendering techniques to overcome this limitation.

The GPGPU technology uses modern graphics cards for computational purposes. The Open Computing Library (OpenCL) [7–10] and Compute Unified Device Architecture (CUDA) [11,12] are representative APIs. Using GPGPU technology, some models have made the existing rasterization pipeline into a fully programmable software graphics pipeline to achieve more flexibility [13–16]. Others have used ray tracing because it can handle massive data with many threads using GPGPU technology. NVIDIA released a ray-tracing engine called OptiX [17,18], which uses the CUDA API, and many researchers have focused on ray tracing while using it. Despite these efforts, existing

computer games or 3D interactive applications are challenging to apply because high-end graphics techniques, such as ray tracing and radiosity, require a massive number of calculations.

Moreover, the CPU overhead was a significant problem in rendering, along with the increased GPU performance. Introducing low-level APIs may be a solution to minimize the overhead. DirectX 12 [5], and Vulkan [2,6,19] are representative APIs. Vulkan supports a variety of platforms, including Windows, Android, SteamOS, and even leading-edge game engines, including Unity [20], Unreal Engine [21,22], and CryEngine [23].

Until now, traditional graphics APIs, such as OpenGL, have been used, and these APIs are moving to Vulkan for efficiency. However, due to the complexity of Vulkan, its use has slowly been increasing. Thus, various attempts have been made to compare Vulkan and OpenGL [24]. This paper aims to construct a framework for *local illumination* using Vulkan.

3 Design of a Graphics Pipeline

The design purpose of Vulkan was to enable detailed control of each function unit in the graphics card; hence, the entire 3D graphics pipeline is divided into relatively small parts (Fig. 1). Traditional OpenGL has relatively large steps (Fig. 2), considering the programmer's convenience. The *Vulkan rasterization pipeline* is based on the original Vulkan pipeline and aims to include all the traditional features of OpenGL.

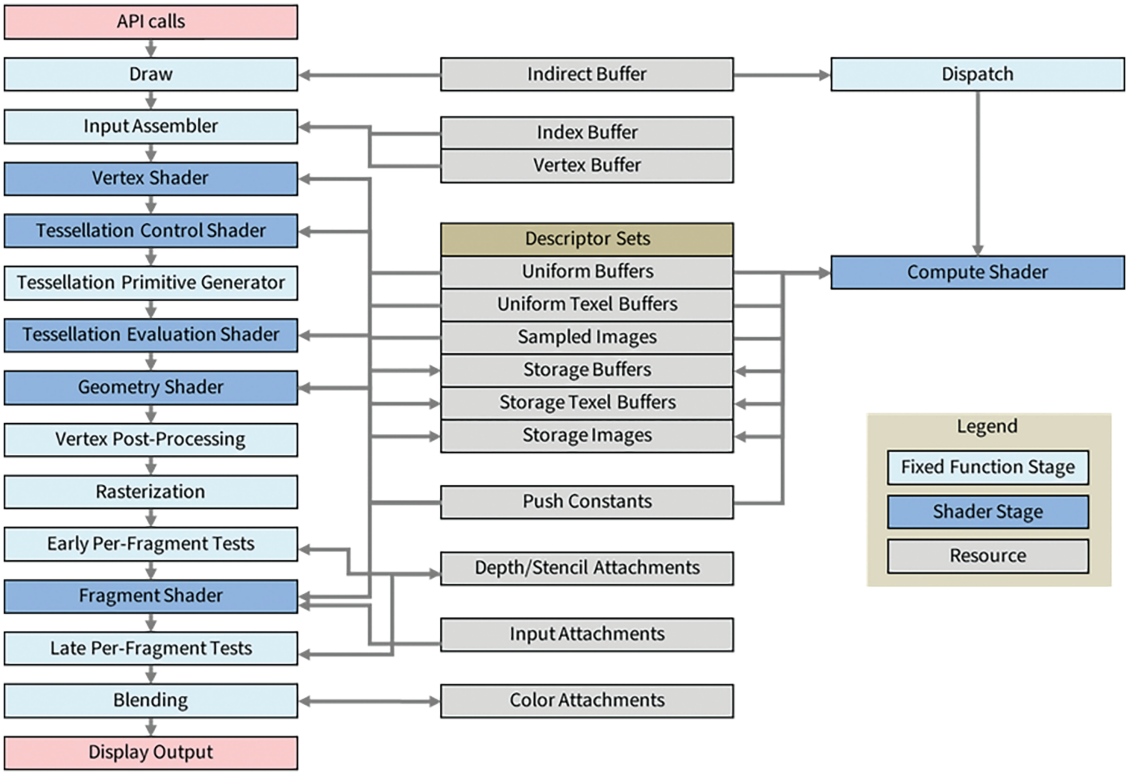


Figure 1: Block diagram of the *Vulkan* 3D graphics pipeline [1]

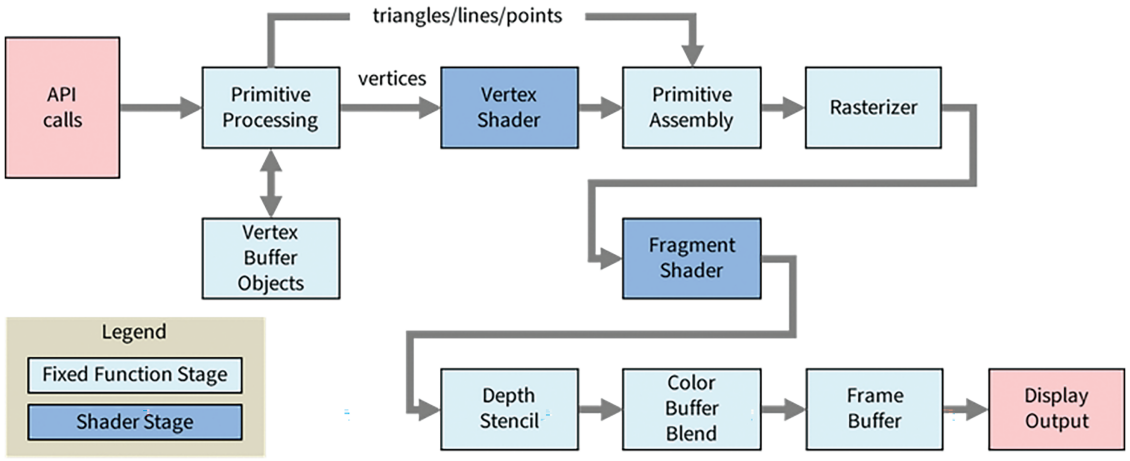


Figure 2: Traditional *OpenGL* 3D graphics pipeline [25]

One of the most explicit differences between Vulkan and OpenGL graphics pipelines is how the *command queue* works. OpenGL creates a *command buffer* for each function and submits it to the physical execution queue in the graphics card. During these processing steps, the underlying graphics card consumes execution time to submit the individual command buffer.

Vulkan uses *batch processing* by preparing the command buffers separately and submitting them all at once. Therefore, even when operating in a single-thread environment, the execution speed of Vulkan can be significantly faster than that of OpenGL. Additionally, Vulkan has the potential for faster execution because it can support multithreading.

The *Standard Portable Intermediate Representation* (SPIR-V) [26] compiling system efficiently uses the Vulkan pipeline. The SPIR-V compilers can hide the kernel source code from the public while providing all the features with SPIR-V byte-code files.

The overall 3D graphics pipeline comprises three steps. The first step is reading the vertex and index data from object files and creating vertex and index buffers. The second step reads the texture and material data and makes *descriptor sets* for each texture with *uniform buffers*. Finally, this approach binds the necessary buffers, push constants, and queue draw calls to command buffers for each sub-mesh.

The overall design was converted to SPIR-V programs and Vulkan API function calls and works as a rasterization framework to render the typical 3D graphics output. The rendering results from this rasterization framework and their analysis are presented in the next section.

4 Implementation Results

4.1 Experimental Environment

This work implements an overall rasterization pipeline with the Vulkan library. A set of typical OpenGL-based animation sequences were selected to check the correctness and efficiency of the Vulkan implementation. From the *Utah 3D Animation Repository* [27] and the *McGuire computer graphics archive* [28], the rasterization pipeline renders four scenes: *fairy forest*, *living room*, *breakfast room*, and *fireplace room* (Fig. 3). These scenes consist of 174, 580, 674, and 143 K triangles.

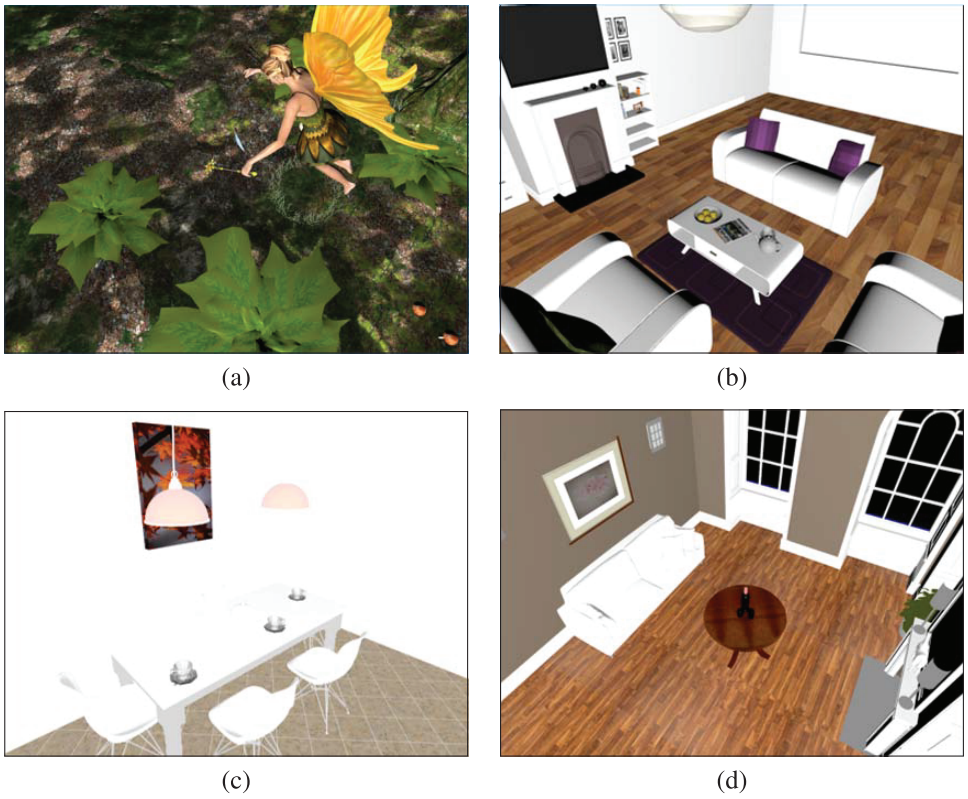
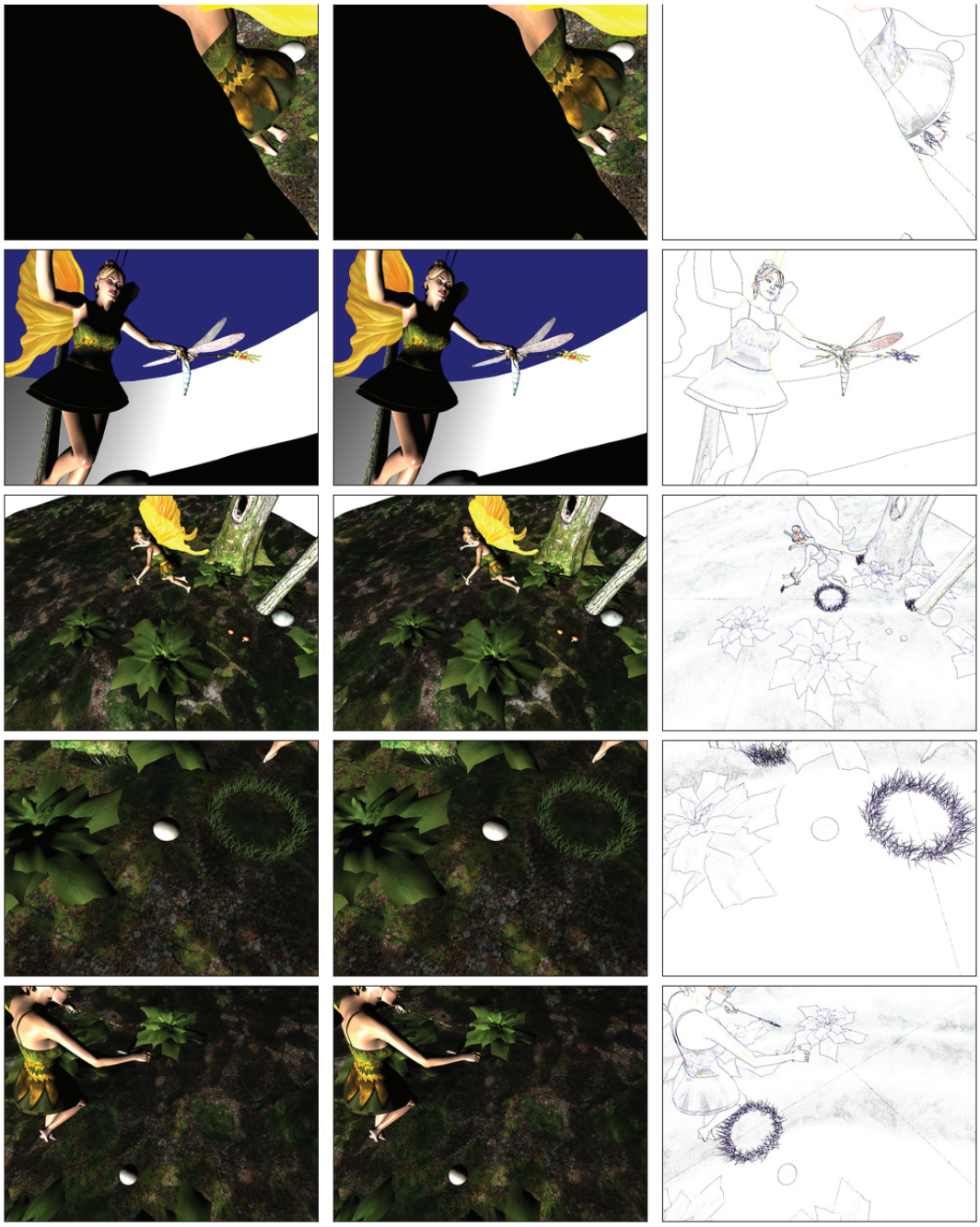


Figure 3: Rendering sequences: (a) *fairy forest*, with public permission from the *Utah 3D Animation Repository* [27]; (b) *living room*, (c) *breakfast room*, (d) *fireplace room*, with CC BY 3.0 permission from the *McGuire Computer Graphics Archive* [28]

These scenes were originally designed to be rendered with OpenGL; hence, these scenes can be rendered with the OpenGL library. These OpenGL rendering results are used as the correct results to check the correctness of the implementation. The original OpenGL version and new Vulkan rasterization were implemented on the same hardware platform, a Windows 11 computer with AMD Ryzen 7, 8-core 3.70 GHz CPU with two graphics cards: Radeon RX Vega 56 and NVIDIA RTX 2080.

4.2 Proof of Correctness

The scene of the "fairy forest" in Fig. 4 is rendered using the Vulkan rasterization pipeline, as presented in the left column of Fig. 4a. Then, it is re-rendered with the OpenGL functionalities, as depicted in the middle column of Fig. 4b. The differences between these two renderings are presented in the right column of Fig. 4c to check the correctness of the Vulkan rendering concerning the traditional OpenGL rendering.



(a) (b) (c)

Figure 4: Rendering results for the *fairy forest*: (a) proposed results, (b) OpenGL results, (c) differences ($\times 32$)

In the early stages of our experiments, we found some unexpected mismatches on these pixel-by-pixel comparisons. Most of those differences were found in the textured regions, and we carefully arranged the texture parameters for both graphics pipelines to the following values:

- texture magnification filter: linear
- texture minification filter: linear
- texture mip-map mode: all linear
- texture address mode: all repeat
- texture anisotropic filter: disabled
- unnormalized texture coordinates: disabled.

After these arrangements, most differences are found at the object boundaries. Since graphics pipelines use a large amount of floating-point number computations especially for coordinate transformations, we cannot completely avoid accumulated numerical differences between different graphics pipelines, and it results in the slightly misaligned pixel positions at the object boundaries.

The rendered images are stored in the typical RGB format, where each pixel is represented in three color tuples (red, green, and blue). Numerical values in these color tuples are expressed as 8-bit integer values $[0, 255]$. Thus, the difference images in Fig. 4c may have pixel values from 0 to 255. However, the experiment reveals that most actual differences were small enough to be much smaller than even 32. Therefore, the difference images were magnified for convenience with a scale factor of 32. Finally, white pixels indicate no error (difference 0), whereas black pixels present a difference value of 32. Gray pixels are in-between values of the pixel differences.

The same renderings were conducted for the other drawing scenes of the "living room", "breakfast room", and "fireplace room", taking their differences, as presented in Figs. 5–7, respectively. The differences between these renderings were analyzed theoretically. Table 1 presents the mean squared error (MSE), root mean squared error (RMSE), peak signal-to-noise ratio (PSNR), and structural similarity index measure (SSIM) values [29,30].

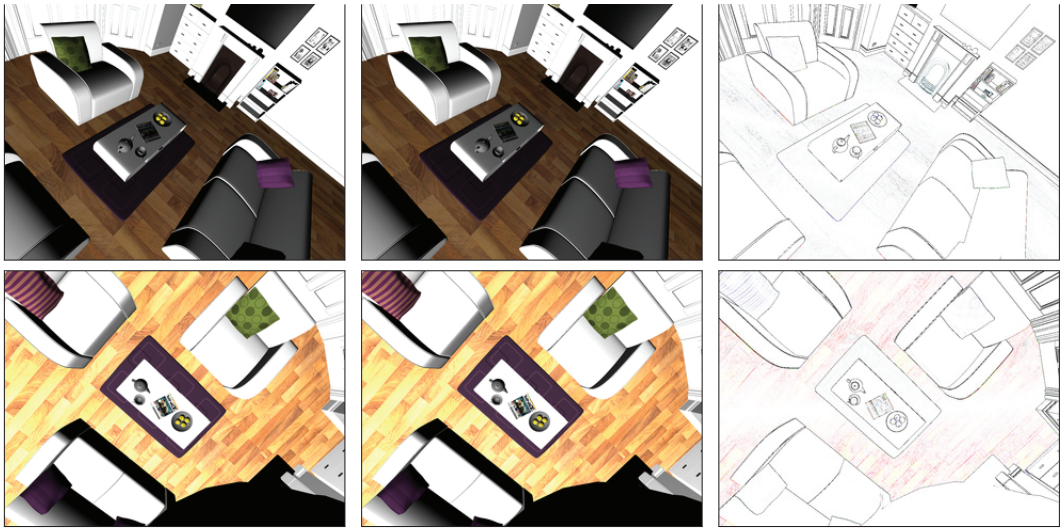
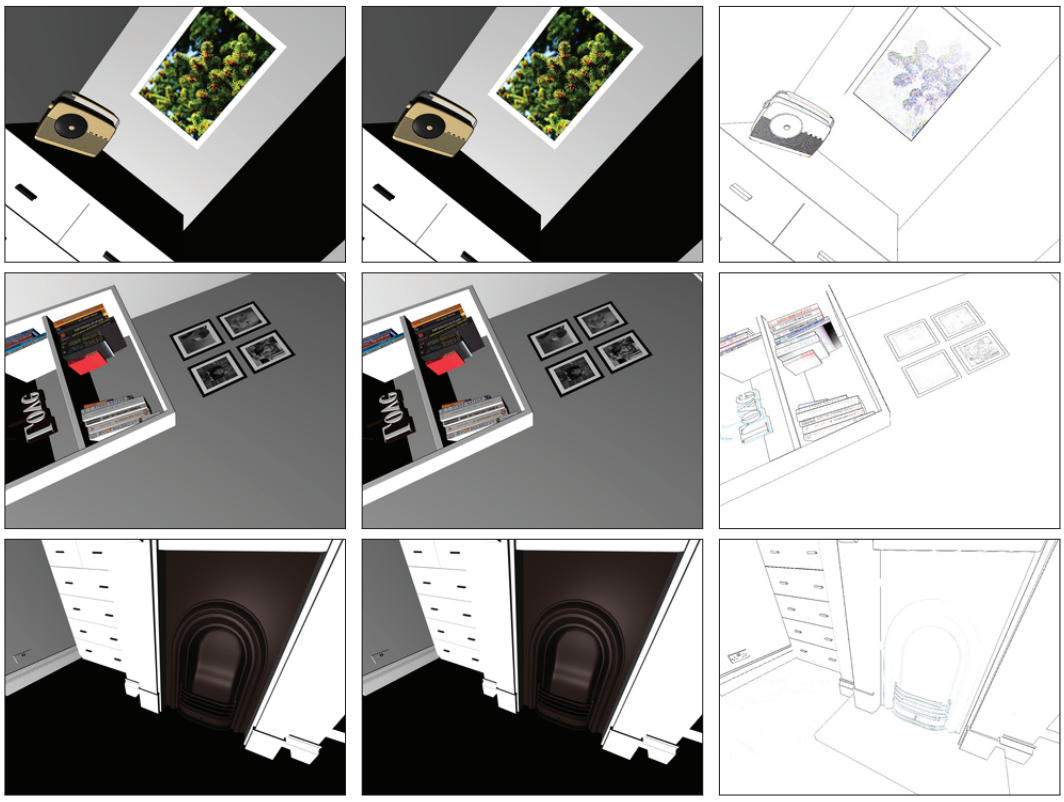


Figure 5: (Continued)



(a) (b) (c)

Figure 5: Rendering results for the *living room*: (a) proposed results, (b) OpenGL results, (c) differences ($\times 32$)

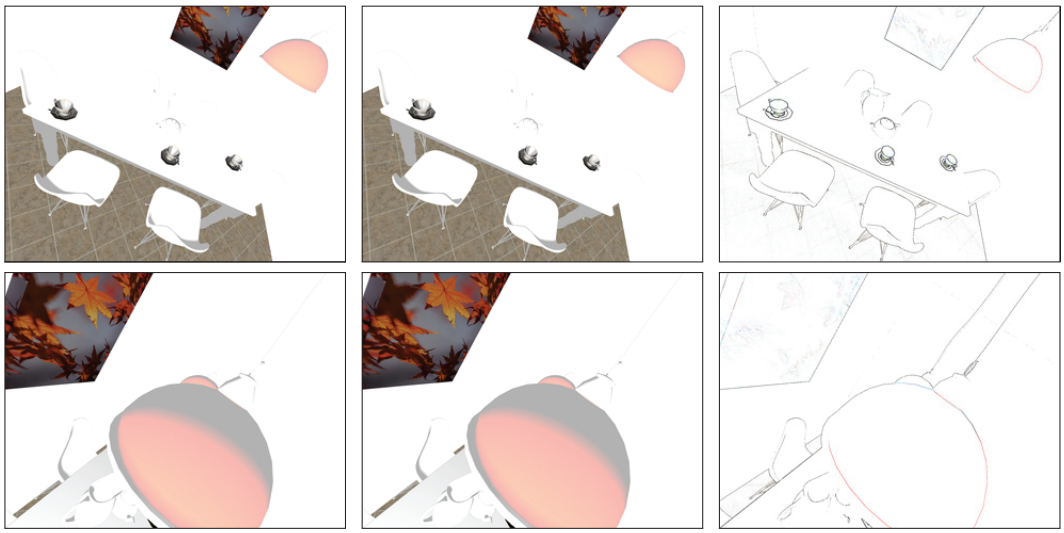
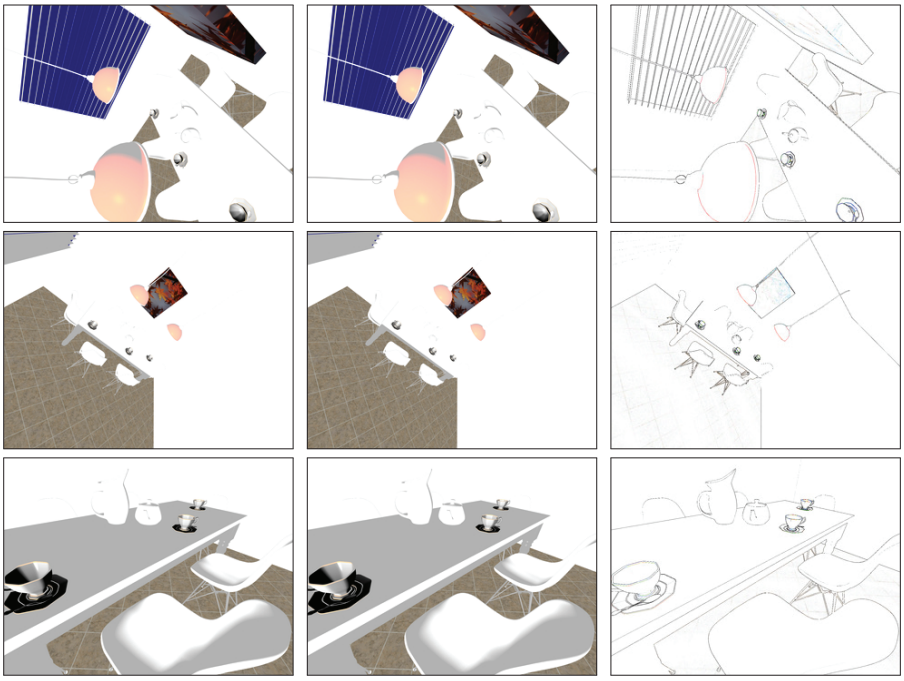


Figure 6: (Continued)



(a) (b) (c)

Figure 6: Rendering results for the *breakfast room*: (a) proposed results, (b) OpenGL results, (c) differences ($\times 32$)

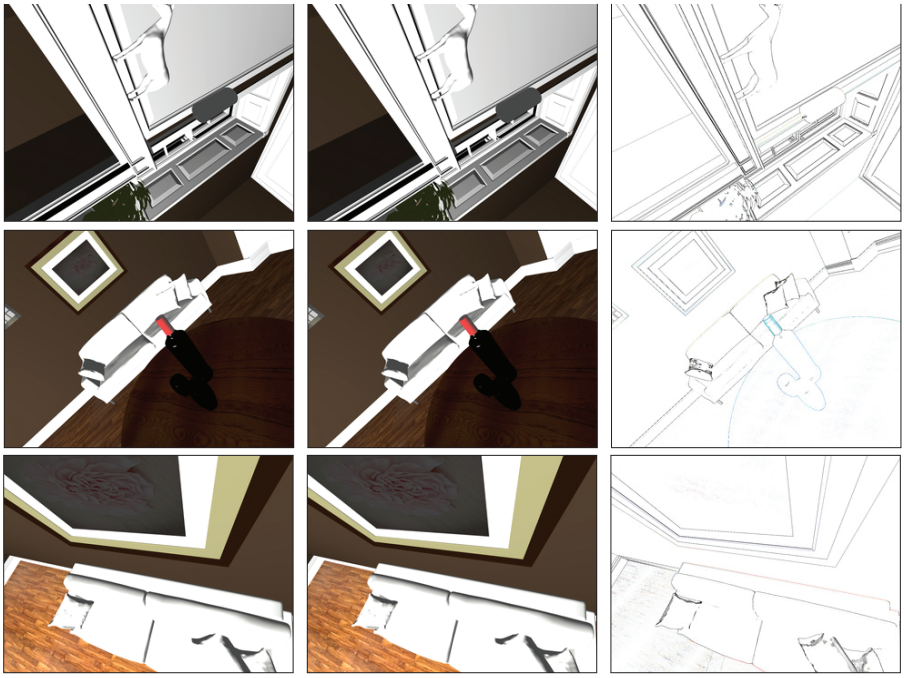
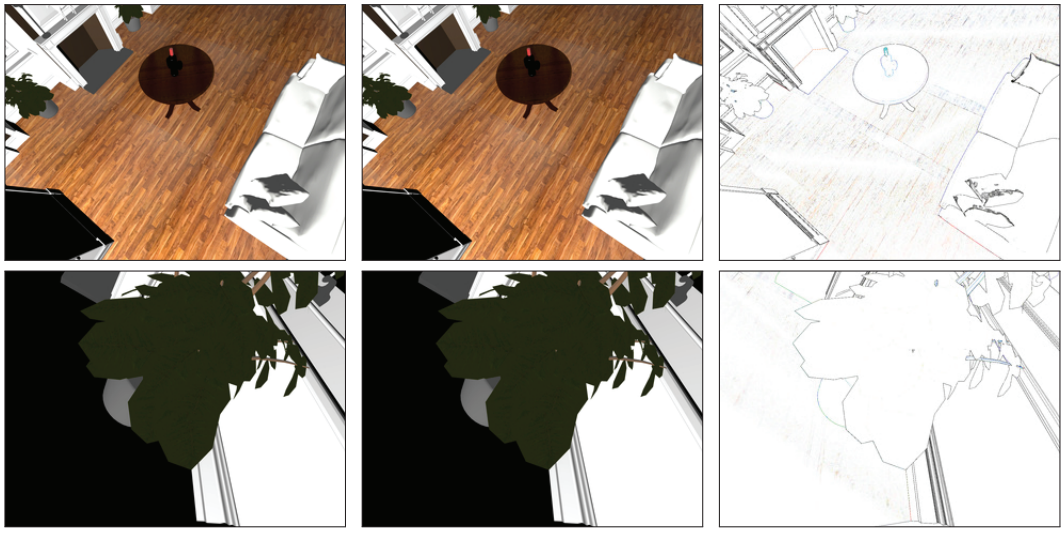


Figure 7: (Continued)



(a) (b) (c)

Figure 7: Rendering results for the *fireplace room*: (a) proposed results, (b) OpenGL results, (c) differences (×32)

Table 1: Rendering exactness: error values

	MSE[1]	RMSE[2]	PSNR[3]	SSIM[4]
Fairy forest	1.915	1.384	42.486	0.999
Living room	1.790	1.338	43.075	0.999
Breakfast room	0.428	0.654	55.509	0.999
Fireplace room	0.690	0.831	51.353	0.999
Average	1.206	1.052	48.106	0.999

Note: [1] mean square error; [2] root mean square error; [3] peak signal-to-noise ratio; [4] structural similiarity index measure.

Each red, green, and blue component of the images was evaluated separately, and the MSE was calculated by measuring the average of these. Then, the errors are presented in an image format as the difference images, allowing a check of where the errors mainly occur. Table 1 presents the results of comparing the image quality numerically. The PSNR values are about 50, and the SSIM values are about 0.999 or almost 1.0. Although these RMSE and PSNR values indicate some slight errors exist, the values of SSIM reveal that the two images are structurally the same because the upper bound of the SSIM is 1.0. Thus, the rendering results and OpenGL rendering results can be practically regarded as the same.

4.3 Comparison of Efficiency

Tables 2 and 3 compare the rendering performance of the proposed Vulkan rasterization and traditional OpenGL. The average and minimum values for the number of rendering *frames per second* (fps) were calculated to measure the rendering speed. In Table 2, the average fps values indicate that the proposed Vulkan rasterization pipeline can render the scenes 1.4136 to 2.6431 times faster than

traditional OpenGL rendering. The minimum fps values range from 1.4317 to 2.5685 times (Table 3). The proposed Vulkan rasterization outperforms the traditional OpenGL-based rendering, which is the reason for using Vulkan rather than OpenGL.

Table 2: Rendering performances: Average number of frames per second. (unit: frames per second)

Number of triangles		Radeon RX Vega 56			NVIDIA RTX 2080		
		OpenGL (a)	Proposed work (b)	Acceleration ratio (b/a)	OpenGL (a)	Proposed work (b)	Acceleration ratio (b/a)
Fairy forest	(174 K)	677.54	1698.69	2.5071	750.23	1804.76	2.4056
Living room	(580 K)	552.81	1285.92	2.3262	599.05	1583.36	2.6431
Breakfast room	(674 K)	388.41	549.05	1.4136	428.01	804.57	1.8798
Fireplace room	(143 K)	1145.29	1781.07	1.5551	1231.33	1991.09	1.6170
Average		691.01	1328.68	1.9228	752.16	1545.95	2.0554

Table 3: Rendering performances: Minimum number of frames per second. (unit: frames per second)

Number of triangles		Radeon RX Vega 56			NVIDIA RTX 2080		
		OpenGL (a)	Proposed work (b)	Acceleration ratio (b/a)	OpenGL (a)	Proposed work (b)	Acceleration ratio (b/a)
Fairy forest	(174 K)	639.94	1479.55	2.3120	723.59	1518.08	2.0980
Living room	(580 K)	519.24	1041.23	2.0053	529.32	1359.56	2.5685
Breakfast room	(674 K)	349.69	500.64	1.4317	395.40	695.09	1.7579
Fireplace room	(143 K)	1098.52	1617.17	1.4721	1115.78	1619.61	1.4515
Average		651.85	1159.65	1.7790	691.02	1298.09	1.8785

Introducing the new Vulkan standard, they emphasized that the new graphics library will reduce the CPU loads. In our experiments, we also compared CPU and memory usages using the system query API functions provided by the Windows operating system [31]. CPU usage was measured by averaging the percentage of total CPU time spent on both of traditional OpenGL rendering and proposed Vulkan rasterization. As shown in Table 4, CPU usage is reduced to at best 42% to at worst same to the traditional OpenGL rendering. It is a noticeable improvement with our Vulkan rasterization, from the viewpoint of the CPU usage.

For memory usage, we measured the actual RAM usage during execution and calculated the average. As shown in Table 5, proposed Vulkan rasterization uses only 45% to 70% of the RAM required by traditional OpenGL rendering. Since OpenGL operates as a state machine, it maintains overall information including the current state, current texture image, and many others. For these structural reasons, OpenGL's memory usage explicitly increases, and this should be one of the reasons for migrating to Vulkan.

Table 4: CPU usage: average percent of CPU load

Number of triangles		Radeon RX Vega 56			NVIDIA RTX 2080		
		OpenGL (a)	Proposed work (b)	Ratio (b/a)	OpenGL (a)	Proposed work (b)	Ratio (b/a)
Fairy forest	(174 K)	15.860%	11.597%	0.731	22.210%	22.377%	1.008
Living% room	(580 K)	15.457%	9.373%	0.606	20.550%	20.113%	0.979
Breakfast room	(674 K)	19.773%	10.303%	0.521	27.020%	18.787%	0.695
Fireplace room	(143 K)	26.077%	11.097%	0.426	25.183%	24.497%	0.973
Average		19.292%	10.593%	0.549	23.741%	21.443%	0.903

Table 5: Memory usage: average RAM requirements. (unit: MB)

Number of triangles		Radeon RX Vega 56			NVIDIA RTX 2080		
		OpenGL (a)	Proposed work (b)	Ratio (b/a)	OpenGL (a)	Proposed work (b)	Ratio (b/a)
Fairy forest	(174 K)	250.102	135.990	0.544	218.145	153.691	0.705
Living% room	(580 K)	300.733	180.849	0.601	301.153	206.200	0.685
Breakfast room	(674 K)	240.549	126.753	0.527	248.339	150.178	0.605
Fireplace room	(143 K)	239.677	108.583	0.453	215.680	130.782	0.606
Average		257.765	138.044	0.536	245.829	160.213	0.652

As a simple measure, the number of source code lines for both rendering implementations was counted for the implementation cost, as listed in Table 6. The proposed Vulkan rasterization pipeline requires 2.211 times more source code lines than the traditional OpenGL renderings. Vulkan is a low-level library; hence, every detail of the device driver settings is needed, and the final source codes are verbose. As demonstrated, the implementation cost is a weak point of Vulkan and should be reduced for broader use. Reducing the source code complexity of proposed Vulkan rasterization would be one of our future works.

Table 6: Implementation cost: comparing the number of source code lines

	The number of source code lines
OpenGL implementation (a)	1102 lines
Proposed work (b)	2437 lines
Ratio (b/a)	2.211 times increased

5 Conclusions and Future Work

This paper presents examples of configuring a typical 3D graphics pipeline using Vulkan, revealing its effectiveness by comparing it with OpenGL implementations. The experimental results demonstrate

that the Vulkan implementation worked *much faster and more effectively* than the OpenGL implementation. However, the number of source code lines increased considerably. A set of repeatedly used Vulkan operations must be abstracted and simplified for efficient programming.

In the future, we will improve the proposed framework for various uses, including ray-tracing effects. Ray tracing techniques were previously possible to implement using *compute shaders* in traditional OpenGL rendering [3,4]. With the introduction of Vulkan *ray tracing extensions* [32–34], API-level ray tracing development is now possible. In particular, Vulkan offers accelerated graphics data structures like bounding volume hierarchy (BVH) and a more advanced shader architectures than compute shaders, enabling more efficient implementations. Our rasterization framework would be integrated with these extensions, particularly hardware acceleration including NVIDIA RT cores [35] and Radeon RDNA2 [36], to integrate ray tracing effects into our Vulkan rasterization pipeline. This integration would enable more realistic effects with relatively efficient rendering efforts, as one of our explicit future works.

Another item for our consideration will be the *mobile graphics* environment. This paper focused on the Vulkan rasterization in the desktop environment, and extension to the mobile environment is also necessary. The unified memory architecture (UMA) can exhibit completely different characteristics in terms of memory management, requiring additional experiments and analysis. Considering the mobile graphics processor architectures and their emphasis on texture compression techniques, more experiments and analyses for those configurations are necessary.

Acknowledgement: Not applicable.

Funding Statement: This work was supported by the IITP (Institute of Information & Communications Technology Planning & Evaluation)—ITRC (Information Technology Research Center) grant funded by the Korea government (Ministry of Science and ICT) (IITP-2025-RS-2024-00437756, 80%). 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 (41202420214871, 10%). This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MIST) (RS-2023-00242528, 10%).

Author Contributions: The authors confirm contribution to the paper as follows: study conception and first-stage design: Mingyu Kim, Nakhoon Baek; data collection: Mingyu Kim; analysis and interpretation of results: Mingyu Kim, Nakhoon Baek; draft manuscript preparation: Mingyu Kim, Nakhoon Baek; manuscript revise: Nakhoon Baek. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The datasets generated during or analyzed during the current study are available from the corresponding author on reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Khronos Vulkan Working Group. Vulkan specification, version 1.4.323. Beaverton, OR, USA: Khronos Group; 2025.
2. Sellers G, Kessenich J. Vulkan programming guide: the official guide to learning vulkan. 1st ed. Boston, MA, USA: Addison-Wesley Professional; 2016.
3. Segal M, Akeley K. The OpenGL graphics system: a specification, version 4.6 (core profile). Beaverton, OR, USA: Khronos Group; 2022.
4. Gordon VS, Clevenger J. Computer graphics programming in OpenGL with C++. 1st ed. Herndon, VA, USA: Mercury Learning and Information; 2024.
5. Luna F. Introduction to 3D game programming with DirectX 12. Herndon, VA, USA: Mercury Learning and Information; 2016.
6. Bailey M. The Vulkan computer graphics API. In: ACM SIGGRAPH, 2023 courses. New York, NY, USA: Association for Computing Machinery; 2023. doi:10.1145/3587423.3595529.
7. Munshi A, Gaster B, Mattson TG, Ginsburg D. OpenCL programming guide. Boston, MA, USA: Addison-Wesley; 2011.
8. Khronos OpenCL Working Group. The OpenCL specification, version 1.2. Beaverton, OR, USA: Khronos Group; 2012.
9. Khronos OpenCL Working Group. The OpenCL specification, version 2.2. Beaverton, OR, USA: Khronos Group; 2019.
10. Khronos OpenCL Working Group. The OpenCL specification, version 3.0.19. Beaverton, OR, USA: Khrono Group; 2025.
11. Sanders J, Kandrot E. CUDA by example: an introduction to general-purpose GPU programming. Boston, MA, USA: Addison-Wesley; 2011.
12. NVIDIA. CUDA toolkit documentation, version 13.0, 2025 [Internet]. [cited 2025 Oct 1]. Available from: https://docs.nvidia.com/cuda/.
13. Kenzel M, Kerbl B, Schmalstieg D, Steinberger M. A high-performance software graphics pipeline architecture for the GPU. ACM Trans Graph. 2018;37(4):140. doi:10.1145/3197517.3201374.
14. Laine S, Karras T. High-performance software rasterization on GPUs. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. New York, NY, USA: Association for Computing Machinery; 2011. p. 79–88.
15. Liu F, Huang M-C, Liu X-H, Wu E-H. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. New York, NY, USA: Association for Computing Machinery; 2010. p. 75–82.
16. Kim M, Baek N. A 3D graphics rendering pipeline implementation based on the OpenCL massively parallel processing. J Supercomput. 2021;77(7):7351–67. doi:10.1007/s11227-020-03581-8.
17. Parker SG, Bigler J, Dietrich A, Friedrich H, Hoberock J, Luebke D, et al. OptiX: a general purpose ray tracing engine. ACM Trans Graph. 2010;29(4):1–13.
18. Ludvigsen H, Elster AC. Real-time ray tracing using NVIDIA OptiX. In: 31st Annual Conference of the European Association for Computer Graphics, Eurographics 2010—Tutorials; 2010 May 3–7; Norrköping, Sweden. p. 65–8.
19. Zhang A, Chen K, Johan H, Erdt M. High performance city rendering in Vulkan. In: SIGGRAPH Asia 2018 Posters. New York, NY, USA: Association for Computing Machinery; 2018. doi:10.1145/3283289.3283342.
20. Suvak J. Learn Unity3D programming with unityscript: unity's javascript for beginners. New York, NY, USA: Apress; 2014.
21. Moniem MA. Mastering unreal engine 4.X. Birmingham, UK: Packt Publishing; 2016.
22. Sanders A. An introduction to unreal engine 4. Boca Raton, FL, USA: Taylor & Francis, CRC Press; 2016.

23. Hussain F, Hussain K. CryEngine basics: first steps in game development. Geneva, Switzerland: Sonar Publishing; 2024.
24. Ferraz O, Menezes P, Silva V, Falcao G. Benchmarking vulkan vs OpenGL rendering on low-power edge GPUs. In: 2021 International Conference on Graphics and Interaction (ICGI); 2021 Nov 4–5; Porto, Portugal. p. 1–8.
25. Segal M, Akeley K. The OpenGL graphics system: a specification, version 3.3. Beaverton, OR, USA: Khronos Group; 2010.
26. Khronos S.P.I.R.Working Group. SPIR-V specification, version 1.6, revision 6. Beaverton, OR, USA: Khronos Group; 2025.
27. Ingo Wald. Utah 3D animation repository. [Internet]. 2025 [cited 2025 Oct 1]. Available from: https://www.sci.utah.edu/.
28. McGuire M. Computer graphics archive. [Internet]. 2017 Jul [cited 2025 Oct 1]. Available from: https://casual-effects.com/data/.
29. Horé A, Ziou D. Image quality metrics: PSNR vs. SSIM. In: 2010 20th International Conference on Pattern Recognition; 2010 Aug 23–26; Istanbul, Turkey. p. 2366–9.
30. Huynh-Thu Q, Ghanbari M. Scope of validity of PSNR in image/video quality assessment. Electron Lett. 2008;44:800–1. doi:10.1049/el:20080522.
31. Pavel Yosifovich. Windows kernel programming. Redmond, WA, USA: Microsoft Press; 2021.
32. The Khronos Vulkan working group. Vulkan KHR acceleration structure-device extension. Beaverton, OR, USA: Khronos Group; 2021.
33. The Khronos Vulkan working group. Vulkan KHR ray tracing pipeline-device extension. Beaverton, OR, USA: Khronos Group; 2020.
34. The Khronos Vulkan working group. Vulkan KHR ray query-device extension. Beaverton, OR, USA: Khronos Group; 2020.
35. NVIDIA. NVIDIA Turing GPU Arhictecture [Internet]. 2018 [cited 2025 Oct 1]. Available from: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.
36. Advanced Micro Devices. AMD RDNA performance guide [Internet]. 2024 [cited 2025 Oct 1]. Available from: https://gpuopen.com/learn/rdna-performance-guide/.