Науковий вісник НЛТУ України
Scientific Bulletin of UNFU
ISSN 1994-7836 (print)
ISSN 2519-2477 (online)
https://doi.org/10.36930/40350317
Article received 11.05.2025
Article accepted 29.05.2025
UDC 004.92
Correspondence author: M. Yu. Protsyk maksym.protsyk.mnpzm.2023@lpnu.ua

M. Yu. Protsyk, T. O. Korotyeyeva
Lviv Polytechnic National University, Lviv, Ukraine

ANALYSIS THE EFFICIENCY OF PROGRAMMING TOOLS FOR 3D RENDERING IN VIDEO GAME DEVELOPMENT

Today, with the availability of many rendering APIs, modern developers have much more freedom in choosing the perfect rendering tool for their needs. However, this task is not so simple without knowing the benefits and drawbacks of the mentioned tools. This study provides a detailed analysis and comparison of the performance of popular APIs for three-dimensional rendering. A flexible comparison tool in the form of a simple game engine based on the ECS (Entity-Component-System) architecture was developed for the mentioned purpose. The engine supports configuration through JSON files. Rendering using the following three popular APIs was selected for evaluation: DirectX 11, OpenGL, and Vulkan. For a fair comparison, this research conducts the experiment on two computer configurations with integrated (Intel HD Graphics 520) and discrete (Nvidia GeForce RTX 4060) GPUs. The experimental scenarios included rendering static and dynamic scenes using three types of models of different sizes in amounts up to 120 for integrated GPU and up to 2000 for discrete GPU. For each of the tested scenarios, a set of performance metrics was measured, including the number of rendered frames per second, a load of GPU, GPU memory usage, RAM usage, and others. The results revealed that OpenGL is faster than DirectX and Vulkan when executing experimental scenarios on an integrated graphics processor for all models. At the same time, for a discrete GPU, the outcome unveiled the advantage of Vulkan over the other two APIs for complex models and the superiority of DirectX for simple ones. The mentioned conclusions were confirmed with the application of statistical tests. Moreover, this research has discovered that DirectX is the most efficient in terms of the usage of computer resources. This research may be helpful for developers of video games or other high-performance applications in the field of 3D rendering. Furthermore, the developed game engine allows for the further extension of the list of evaluated APIs in the future.

Keywords: computer graphics; game engine; real-time rendering; performance measurement; static experimental scenario.

Introduction / Вступ

In the modern world of video games, where technologies are developing rapidly, choosing an effective visualization method has become one of the primary tasks in creating a game product. The increase in the power of personal computers has allowed developers to gradually introduce technologies into their games that were impossible to use only a few years ago in this area because most video games require real-time calculations. A speed equal to one or even ten visualized frames per second is too low here [3] because the game in this form will not feel smooth and will not satisfy the end user. With the availability of the latest technologies, such as ray tracing, the gaming industry is entering a new era of graphic realism. Recent studies, such as the publication by Feng Xie et al. [21], demonstrate the value of ray tracing in creating strong and realistic visual effects, and, most importantly, its potential for real-time application. Innovations in rendering are opening new possibilities for game graphics, offering improved visualization and greater immersion in the game world. However, such progress not only opens the door to improving the quality of the visual experience in games but also presents developers with the difficult task of choosing the most suitable rendering methods and optimizing them [8] for specific projects. It is one thing to display static objects and quite another to display dynamic ones, and the number of these objects plays an equally significant role. Choosing the correct technique and tools cannot be overstated, as they directly impact graphics quality, game performance, and the overall gaming experience.

Object of research – comparing three-dimensional image rendering APIs in video game development.

Subject of research – methods and tools for determining the efficiency of the rendering API for the visualization of three-dimensional images, which will allow architectures and metrics can be used in the research.

The purpose *of the* research – analyze the performance of three-dimensional rendering APIs for game development, which will allow to evaluate the effectiveness of each API when rendering models of different sizes in dynamic and static scenarios on discrete and integrated GPUs.

To achieve this *purpose*, the following main *research objectives are identified*:

- analyze current publications on 3D rendering, its performance, and game engine development to find out, which archi-

Інформація про авторів:
Процик Максим Юрійович, магістрант, кафедра програмного забезпечення.
Email: maksym.protsyk.mnpzm.2023@lpnu.ua; https://orcid.org/0009-0006-5488-6179
Коротєєва Тетяна Олександрівна, канд. техн. наук, доцент, кафедра програмного забезпечення.
Email: tetyana.o.korotyeyeva@lpnu.ua; https://orcid.org/0000-0002-6287-5895
Цитування за ДСТУ: Процик М. Ю., Коротєєва Т. О. Аналіз ефективності роботи програмних засобів для 3D-рендерингу в сфері розроблення відеоігор. Науковий вісник НЛТУ України. 2025, т. 35, № 3. С. 158–165.
Citation APA: Protsyk, M. Yu., & Korotyeyeva, T. O. (2025). Analysis the efficiency of programming tools for 3D rendering in video game development. *Scientific Bulletin of UNFU*, 35(3), 158–165. https://doi.org/10.36930/40350317

tectures and metrics can be used in the research;
- develop an application in the form of a basic game engine that allows the replacement of the rendering method easily;
- implement rendering based on DirectX 11, OpenGL, and Vulkan into the application, which will allow to execute experimental scenarios and collect performance metrics for each of the API;
- prepare experimental game scenarios, collect performance statistics, and conduct a detailed analysis of the results, which will provide an opportunity to evaluate the effectiveness and establish best APIs in different scenarios.

Analysis of recent research and publications. Since the study consists of several stages, the game engine and rendering publications should be analyzed separately.

Publications related to game engine architecture. The Entity-Component-System architecture serves as the basis of many modern projects. The advantages of this approach are well described in Toni Harkönen's work [7]. It is worth mentioning that one of the most popular game engines, Unity, uses this architecture. When using ECS, game objects are characterized not by a specific class but by their components. Components of the same type are stored in a continuous chunk of memory, which reduces the number of cache misses. In this way, ECS provides a significant speed-up of interaction with components when working with many objects, unlike the classic approach, which stores the components as the fields of specific game object classes.

In the work of Daniel Götz et al. [5]. The authors emphasize the simple possibility of expanding ready-made projects by third-party developers – creating so-called mods (additional game content that can be downloaded separately from the game and adds new characters, levels, or game modes). Mods, in turn, notably increase the time users spend in the game without requiring additional funds and time from the developers to create the new content. This content is created solely by enthusiasts. The described game engine was made possible by loading the code at runtime and the modularity of the implemented system. The abovementioned approach would be helpful for this work in the future when expanding the list of analysed rendering APIs.

Another example of research on game engines is the work of Carlos Marin et al. [10]. The publication presents an architecture comprising several agents that interact with one another: physics, input/output, rendering, audio, and logic. The authors describe each agent and its interactions with other parts of the system.

Publications related to rendering. In the publication by Lei Xiao et al. [20], the authors focused on the problem of real-time rendering of high-resolution images while achieving a high number of displayed frames per second. They proposed a machine-learning method for increasing the resolution of an image in real time. The described method uses several low-resolution frames to create high-quality reconstructions while reducing blur and restoring fine details. The authors have trained the model on a large artificial dataset obtained from rendering various 3D scenes while moving through them. The experiments showed that the constructed neural network significantly outperforms the results of other scientists in improving resolution and smoothing, both visually and in terms of numerical metrics.

Markus Schütz et al. [16] propose a new pipeline structure for rendering point clouds, which can display up to two billion points while achieving a speed of 60 processed frames per second. The authors note the performance improvement compared to existing approaches, which became possible due to special optimizations, including point grouping, the use of levels of detail, ignoring of invisible points, adaptive precision of point coordinates, and others.

Another interesting publication is the work of Marti Anglada et al. [1]. The authors presented a new microarchitectural approach to optimize the rendering process in GPUs, which uses calculations based on dividing the entire screen into smaller cells. They found that in many cases, a significant number of cells on the screen remain unchanged for several frames. The approach presented in the study is to detect these unchanged cells at one of the early stages of the rendering pipeline. Suppose a cell does not change during two frames in a row. In that case, the pipeline processes it only once, significantly reducing the processing time of one frame and the GPU's power consumption. The authors demonstrated that the described algorithm, on average, accelerates the rendering process by 1.74 times and reduces power consumption by 43 % compared to standard methods. They also showed that the approach is practical in various usage scenarios, including games and other graphical applications.

As can be seen, modern publications related to rendering primarily focus on the optimization and improvement of existing methods, as well as their comparison. When studying articles related to the evaluation of rendering performance, it was discovered that the most popular metrics are FPS (Frames per second) [6, 8, 15, 17, 20] – the number of frames processed per second, PPS (Points per second) [15, 16, 17] – the maximum number of points displayed on the screen per second, GPU memory consumption [8, 17], and frame time [1, 13, 15, 16, 20]. In addition, the amount of RAM used, load on the processor, and GPU can be measured, as was done in the publication of Filip Popovsky et al. [13].

According to the performed literature analysis, there is no general approach to the analysis and comparison of the rendering APIs. Each of the studies either ignores metrics related to recourse usage of the computer, conducts experiments only on one scenario, or uses one computer system (without considering that APIs can work better on different systems). In addition, none of the works used statistical methods to ensure the correctness of the results. That is why this work is needed to establish a pipeline for analysis of the rendering APIs and, more specifically, compare DirectX, Vulkan, and OpenGL.

Research results and their discussion / Результати дослідження та їх обговорення

This section of the study describes the architecture of the implemented pipeline for unbiased analysis of the rendering APIs, the collected metrics, and the details of the conducted experiment. Finally, the obtained statistical data on the performance of each method is evaluated to establish the best APIs in considered scenarios regarding speed and resource usage.

Engine for the experiment. ECS architecture implemented in C++ programming language was selected as the basis for the experiments framework due to its performance advantages. Four managers drive the engine's logic by controlling entities, components, systems, and events.

The entity manager is the simplest. It keeps track of the occupied entity identifiers and allows the creation or destruction of entities.

The components manager is responsible for all components of the entities. It provides methods for retrieving all entities containing specified component types and methods for adding components to a specific entity or deleting them. For convenient storage and quick access to components, the manager uses a sparse set [2] (each component type is stored in a separate sparse set container). This data structure allows checking the presence of a component in an entity and accessing it in O(1) time.

The systems manager is responsible for all the systems that contain the game logic. It allows adding or removing the systems and executing their logic.

The events manager performs all intermediate interactions between systems since they do not have direct access to each other. An example of an event can be a keystroke initiated by the user, a request to terminate the program, and others. In the code, an event can be an object of any class, which stores the information needed to pass between systems.

Configuration system. The engine supports a JSON-based configuration for simple creation and execution of experimental scenarios. The configuration file contains three main fields:

1) Prefabs: this field contains a list of template objects with defined names and components. They can be used later to create entities dynamically or as a base for entities defined in the configuration file.
2) Entities: this field contains a list of objects to create at the start of the scenario's execution. Every entity may contain its list of components and the base prefab name.
3) Systems: this field contains a list of systems to create at the start of the scenario's execution and the parameters of the systems.

The components and systems are configured automatically without requiring specific JSON parsing logic for each class. The programmer only needs to specify serializable components and systems and their serializable properties using already implemented compiler macros.

Implemented rendering methods. Three separate classes perform rendering using the selected APIs: DirectX 11, OpenGL 4.6, and Vulkan (SDK 1.4.304). All of them implement the same *IRenderer* interface and support models in *obj* format with materials that include the following parameters: texture, diffuse color, specular color, ambient color and shininess.

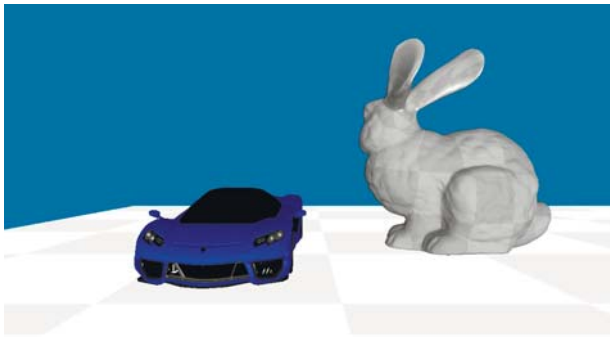

Figure 1. Example of an image rendered by the program (using Vulkan) / Приклад вихідного зображення створеної програми (за використання Vulkan)

For each rendering method, two shaders were implemented, written in the corresponding shader language, repeating the same functionality. The first shader, the vertex shader, receives as input the position of a vertex of the model, its normal, and the coordinate of the texture to use. This shader outputs the transformed position of the vertex. In our case, it performs linear transformations to account for the object's movement and the camera's position. The second shader, the fragment shader, receives as input the normal, the coordinate of the texture, and the transformed position of the vertex obtained from the first shader. It applies the properties of the model's material to the vertex and returns the point's color as output. Fig. 1 shows an example of the image rendered by the program.

Specifications of used computer systems. For a fair comparison, the experimental scenarios were executed on two computers, one with an integrated GPU and the other with a discrete GPU.

First system specifications:

- CPU: Intel Core i5-6200U;
- RAM: 8GB, 2133 MHz;
- GPU: Intel HD Graphics 520;
- OS: Windows 11.

Second system specifications:

- CPU: Intel Core i5-12400F;
- RAM: 32GB, 3200 MHz;
- GPU: Nvidia GeForce RTX4060;
- OS: Windows 11.

Experimental scenarios. Two additional implemented systems allow the execution of static and dynamic scenarios. For both scenarios, the user can specify the model and the number of models to use. The list of models used in the experiment contains three models of different complexity: a cube (12 faces), a teapot (15704 faces), and a bunny (144046 faces) in amounts up to 120 for integrated GPU and up to 2000 for discrete GPU. These numbers were chosen to achieve similar performance for the most complex scenario on both computers. The resolution of rendered images was 1280×720 pixels for all cases.

In the static scenario, the camera and the objects are still. All the displayed objects form a shape similar to a parallelepiped. One side of the parallelepiped is a 10 by 10 square (the user can change the size in the configuration file), and the others expand as the number of objects increases. Additionally, the user can specify the distance between models. Fig. 2 shows an example of this shape for 1200 models.

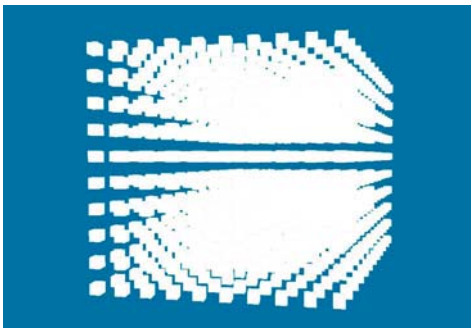

Figure 2. Static experimental scenario for 1200 cube models / Статичний експериментальний сценарій для 1200 моделей куба

In the dynamic scenario, the models form four circles and rotate around the center of the coordinates. The camera also moves back and forth.

The configuration file allows the user to adjust the number of circles and their radii, as well as the rotation speed and camera movement speed. Fig. 3 shows an example of a dynamic scenario for 200 models.

Collected statistical data. A *StatsSystem* class was implemented to collect data on the performance of various rendering methods. It records the following data:

- Average frame time, s;
- Average FPS;
- 99th percentile of frame time, s;
- 1st percentile of frame, s;
- Average, minimum, and maximum amount of RAM used, MB;
- Average, minimum, and maximum amount of GPU memory used, MB;
- Average, minimum, and maximum CPU utilization, %;
- Average, minimum, and maximum GPU utilization, %.

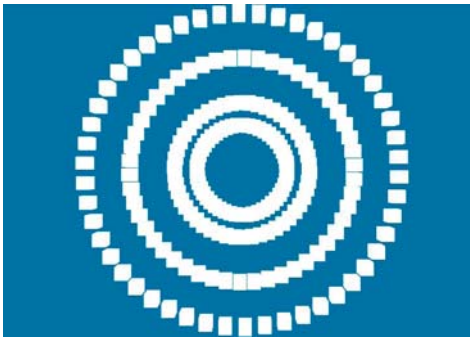

Figure 3. Dynamic experimental scenario for 200 cube models / Динамічний експериментальний сценарій для 200 моделей куба

Obtained research results. Let us discuss the statistical data collected, starting with the computer resources used. Here and later, the focus is solely on static experimental scenarios, as the results are similar to those of the dynamic ones, except for a slight performance increase in all cases. As can be seen from Fig. 4, the amount of RAM appears to depend linearly on the number of objects. At the same time, the observations have proven that the amount of GPU memory increases for Vulkan and remains almost unchanged for OpenGL and DirectX. The constant GPU memory usage is justified by the fact that only unique models are stored in GPU memory. Besides the models, their linear transformation matrices are stored in GPU memory, which is where the difference between Vulkan and the other two APIs lies. In the case of Vulkan, all matrices should remain in memory simultaneously due to its parallelized architecture. In contrast, for the other two rendering APIs, only one matrix is in GPU memory at a time. So, regarding memory usage, Vulkan is the most resource-intensive, and DirectX is the most efficient.

Regarding CPU and GPU utilization, all methods behave similarly to each other when working with the bunny model due to its large size, which is almost 14 MB. In such cases, GPU utilization ranges from 85 % to 95 %, while CPU utilization is lower than 15 % (in this scenario, the GPU becomes a bottleneck). For the cube model, GPU utilization is significantly lower for both OpenGL and DirectX, around 4 % and 20 %, respectively. In comparison, Vulkan still uses around 80 % of its resources. Nevertheless, the increased frame rate increases CPU utilization for all the APIs.

Now, let us discuss the frame rendering time, the primary metric of rendering performance. Fig. 5 displays the average frame time for the discrete GPU. It is easy to see that the results for different models are entirely different. As the complexity of the model increases, the DirectX renderer shows worse results than Vulkan and OpenGL. In the case of the cube model, the best renderer is DirectX, the second is Vulkan, and the last is OpenGL, while for the bunny model, the best renderer is Vulkan, the second is OpenGL, and the worst is DirectX. From this data, it is logical to assume that Vulkan works best when the GPU is a system bottleneck, and DirectX works best when it is not.

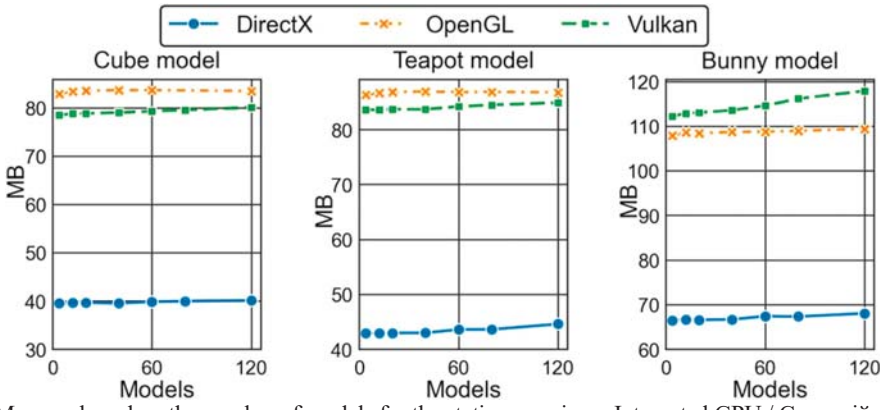

Figure 4. Average RAM usage based on the number of models for the static scenario on Integrated GPU / Середній об'єм використаної оперативної пам'яті залежно від кількості моделей для статичного сценарію на інтегрованій відеокарті

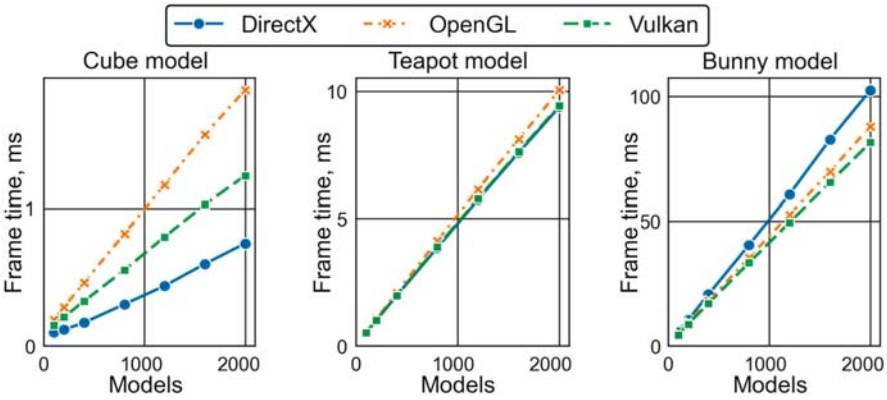

Figure 5. Average frame time based on the number of models for the static scenario on Discrete GPU / Середня тривалість оброблення кадру залежно від кількості моделей для статичного сценарію на дискретній відеокарті

Fig. 6 shows the average frame time for an integrated GPU, and the results are different compared to a discrete GPU. Here, the fastest renderer in three cases is OpenGL. Vulkan and DirectX yield almost identical results for the cube model, but Vulkan is noticeably faster for the other two models. This happens because Vulkan is more suitable for parallel systems, and integrated GPUs are far less parallel than discrete ones.

Another important statistic is the 99th percentile of frame time. It demonstrates how good the performance is in the worst 1 % of cases. For the discrete GPU, the placement of the renderers is the same as for the average frame time. However, this is not the case for the integrated GPU, as seen in Fig. 7. In all cases, the worst renderer is Vulkan, which is not the worst in average frame time. Again, this happens because Vukan is more suitable for parallel and faster GPUs.

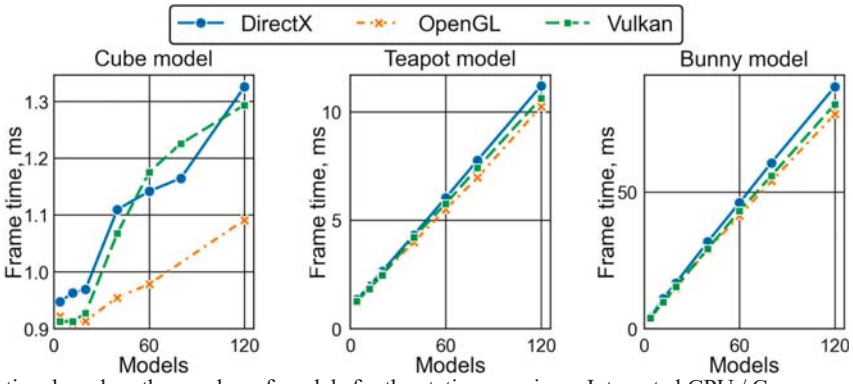

Figure 6. Average frame time based on the number of models for the static scenario on Integrated GPU / Середня тривалість оброблення кадру залежно від кількості моделей для статичного сценарію на інтегрованій відеокарті

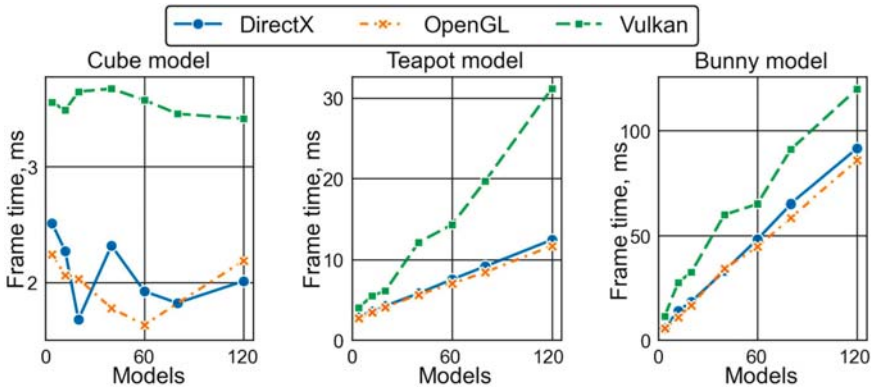

Figure 7. 99th percentile of frame time based on the number of objects for the static scenario on Integrated GPU / 99-ий процентиль часу оброблення одного кадру залежно від кількості моделей для статичного сценарію на інтегрованій відеокарті

To ensure the correctness of the measured results, an already published approach was used [18]: repeat the most complex scenarios for the cube and bunny models 20 times, remove outliers, and use statistical tests to compare the mean values of their distributions. Here, the study ignores the teapot model because the experiments with it would not bring any valuable data. For the discrete GPU, the results were straightforward because the distributions of average FPS for different rendering methods did not overlap. Table 1 displays the mean number of frames per second for this case.

Table 1. Frames per second for the most complex cases on discrete GPU / Кількість оброблених кадрів за секунду для найскладнішого експерименту на дискретному графічному процесорі

	DirectX	OpenGL	Vulkan
Cube	1353.07	534.72	798.82
Bunny	9.90	11.42	12.25

Table 2. Frames per second for the most complex cases on integrated GPU / Кількість оброблених кадрів за секунду для найскладнішого експерименту на інтегрованому графічному процесорі

	DirectX	OpenGL	Vulkan
Cube	757.15	910.11	772.50
Bunny	11.30	12.61	12.20

The results for the integrated GPU were ambiguous. As shown in Fig. 8, the distributions for DirectX and Vulkan overlap for the cube model, and as can be seen in Fig. 9, the distributions for Vulkan and OpenGL overlap for the bunny model.

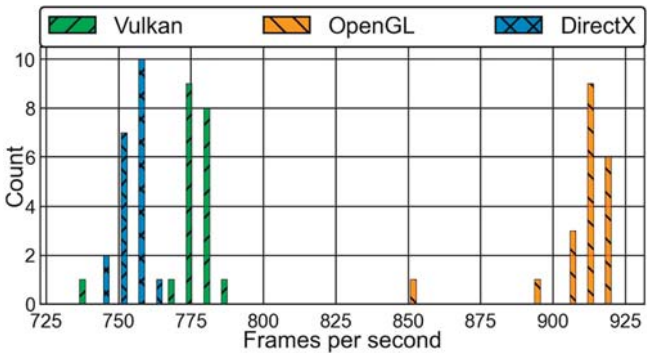

Figure 8. Distribution of FPS for 120 cube models and the static scenario on Integrated GPU / Розподіл середньої кількості оброблених кадрів за секунду для 120 моделей куба в статичному сценарії на інтегрованій відеокарті

The 1.5*IQR* rule [19] removed the outliers for these cases. Here, *IQR* is an interquartile range and equals:

$$IQR = Q_3 - Q_1, \quad (1)$$

where Q_3 is the third quartile of data and Q_1 is the first quartile of the data. All the values X that do not satisfy

$$Q_1 - 1.5IQR \le X \le Q_3 + 1.5IQR, \quad (2)$$

are considered outliers.

After this, the distributions were tested for normality using the D'Agostino-Pearson [4] statistical test. This test has shown that all the distributions of average FPS are normal, allowing us to use the Student's t-test [11] to compare the means of the distributions. For both models, the test results have shown that the mean value of the DirectX average FPS distribution is significantly lower than that of the Vulkan distribution, which in turn is significantly lower than OpenGL's mean. The mean number of frames per second for this case is presented in Table 2.

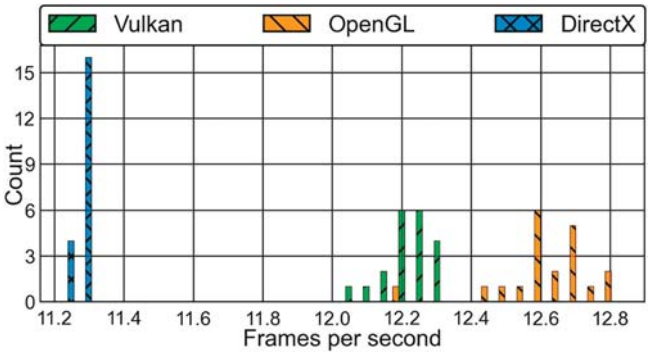

Figure 9. Distribution of FPS for 120 bunny models and the static scenario on Integrated GPU / Розподіл середньої кількості оброблених кадрів за секунду для 2000 моделей зайця у статичному сценарії на інтегрованій відеокарті

Summary and recommendations. Based on the measured results, the study has drawn the following conclusions:

1) For all tested models and the integrated GPU, the fastest rendering API is OpenGL, Vulkan takes second place, and DirectX is the slowest. For the least complex model (cube), OpenGL was 17.8 % faster than Vulkan and 20.2 % faster than DirectX. For the most complex model (bunny), OpenGL was 3.3 % faster than Vulkan and 11.6 % faster than DirectX. It is also worth mentioning that Vulkan has the highest 99th percentile of frame time, which may cause lags more frequently compared to other rendering APIs that provide a more consistent frame rate.
2) For complex models and the discrete GPU, Vulkan is the fastest rendering API. OpenGL is second, and DirectX is the slowest. When rendering the bunny model, Vulkan has shown a 7.3 % and 23.7 % speed increase compared to OpenGL and DirectX.
3) For simple models and the discrete GPU, the fastest rendering API is DirectX, Vulkan takes second place, and OpenGL is the slowest. DirectX has shown a 70 % speed increase compared to Vulkan and was more than 2.5 times faster than OpenGL when rendering the cube model.
4) Regarding RAM usage, DirectX is the most efficient, while Vulkan starts in second place but uses more memory when the complexity and number of models increase. This property makes Vulkan worse than OpenGL for cases with many complex models.
5) Regarding GPU memory usage, the best rendering API is DirectX, the second one is OpenGL, and the worst is Vulkan.
6) Implementation-wise, Vulkan requires much more work because it is low-level, and DirectX and OpenGL are high-level. The implementation difficulty should be considered, even in cases where Vulkan is the fastest, as it may not be the best option due to an extended development time.

Summarizing all of the above, when working with low-end computers, it is recommended to use OpenGL if rendering speed is the priority and DirectX in cases with a hard limit on memory usage. For high-end computers, it is recommended to use DirectX for models up to 1 MB. Vulkan should be used for bigger models if rendering speed is prioritized, and OpenGL should be used if lower resource usage and consistent frame rate are preferred.

Discussion of research results. In the publication [9], the authors compare the performance and energy efficiency of Vulkan and OpenGL. The scene used in tests contains several models of orbiting rocks, and the measured metrics are the power usage (of CPU and GPU) and FPS. The experiments have shown the advantage of Vulkan over OpenGL due to its ability to use less energy while maintaining the same frame rate.

The study [14] compares volume ray-casting implementations based on OpenGL, OpenCL, and CUDA. The implementations were tested using three datasets, and the results show that the compute shader implementation using OpenGL is the best in terms of performance. The main drawback of this work is that only one metric was measured – frame time.

In the work [12], the author aims to port a CUDA snow simulation program from OpenGL to Vulkan while comparing performance for both APIs. The FPS was measured separately for different parts of the rendered scene, and the results show that, in some cases, Vulkan can be two times faster than OpenGL.

In the publication [13], the authors evaluate performance of different rendering toolsets in the 3DS Max application: Scanline Rendering, Mental Ray, V-Ray, and Corona Renderer. The rendering efficiency is measured by three metrics: rendering time, RAM usage, and quality of the resulting image. The results provide recommendations on the best toolsets in different cases.

The study [6] aims to compare optimal scanline voxelization algorithms, a subset of rendering algorithms. For this purpose, the author has implemented Real Line Voxelization, Integer Line Voxelization, and a 3D Bresenham line drawing algorithm. The author has measured the rendering time for several models with different rendering resolutions and has shown that Real Line Voxelization is the best regarding this metric.

As can be seen, the published works are all based on comparing other APIs and algorithms (or just two of the three described in this study). They also lack experiments on different GPUs and statistical confirmation of the results.

So, based on the results of the work performed, it is possible to formulate the following scientific novelty and practical significance of the research results.

Scientific novelty of the obtained research results – improved methodology for performance evaluation of rendering APIs through testing on multiple computer configurations, expanding the list of metrics used, and applying statistical tests, which results in unbiased conclusions.

Practical significance of the research results – the results of the study can be used in game development to select the most suitable rendering API for a specific project, and additionally, the implemented evaluation framework can be used to expand the list of analyzed APIs further if required.

Conclusions / Висновки

The performance of three-dimensional rendering APIs for game development was analyzed, which allowed to

evaluate the effectiveness of each API when rendering models of different sizes in dynamic and static scenarios on discrete and integrated GPUs. Based on the results of the research the following main conclusions can be drawn.

1. Recent publications related to game engine architectures and rendering were reviewed. The literature analysis has shown that studies on rendering mainly do not perform a detailed performance evaluation and focus on 2 to 3 performance metrics. According to this, the need for work that proposes a more thorough approach to evaluating performance has been raised.
2. A flexible framework for conducting the experiment based on ECS architecture in the C++ programming language was implemented. This approach provides high performance, which is crucial for comparison of rendering APIs. In addition, it allows for easy creation of experimental scenarios.
3. The three most popular rendering APIs in game development (DirectX, OpenGL, and Vulkan) were integrated into the engine for further collection of performance metrics. Each of the API followed the same logic for rendering *obj* models to minimize bias due to implementation differences.
4. The implemented rendering APIs were tested in two experimental scenarios, with performance metrics collected separately for three types of three-dimensional models of different sizes. The experiment and the further application of statistical tests have allowed for the establishment of detailed recommendations on the best rendering API for different sizes of models and GPU types.

References

1. Anglada, M., de Lucas, E., Parcerisa, J. M., Aragón, J. L., Marcuello, P., & González, A. (2019). Rendering elimination: Early discard of redundant tiles in the graphics pipeline. *2019 IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), pp. 623–634. https://doi.org/10.48550/arXiv.1807.09449
2. Briggs, P., & Torczon, L. (1993). An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems* (*LOPLAS*), *2*(1–4), 59–69. https://doi.org/10.1145/176454.176484
3. Claypool, K. T., & Claypool, M. (2007). On frame rate and player performance in first person shooter games. *Multimedia systems*, *13*(1), 3–17. https://doi.org/10.1007/s00530-007-0081-1
4. D'Agostino, R. B. (2017). *Goodness-of-Fit-Techniques*. London: Taylor and Francis Group. 1st Edition, 576 p. URL: https://www.routledge.com/Goodness-of-Fit-Techniques/DAgostino/p/book/9780367580346?srsltid=AfmBOorQE-Il545MbztcpSke19EBa0opSVP1KK49vb35rdEFhJyBpZGb0
5. Götz, D., & von Mammen, S. (2023). Modulith: A Game Engine Made for Modding. *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 1–8. https://doi.org/10.1145/3582437.3582486
6. Håkansson, T. (2020). A Comparison of Optimal Scanline Voxelization Algorithms. URL: https://www.diva-portal.org/smash/get/diva2:1459330/FULLTEXT01.pdf
7. Härkönen, T. (2019). Advantages and Implementation of Entity-Component-Systems. URL: https://trepo.tuni.fi/bitstream/handle/123456789/27593/H%C3%A4rk%C3%B6nen.pdf
8. Koulaxidis, G., & Xinogalos, S. (2022). Improving mobile game performance with basic optimization techniques in unity. *Modelling*, 3(2), 201–223. https://doi.org/10.3390/modelling3020014
9. Lujan, M., Baum, M., Chen, D., & Zong, Z. (2019). Evaluating the performance and energy efficiency of opengl and vulkan on a graphics rendering server. *2019 International Conference on Computing, Networking and Communications* (*ICNC*), pp. 777–781. https://doi.org/10.1109/ICCNC.2019.8685588
10. Marin, C., Chover, M., & Sotoca, J. M. (2019). Prototyping a game engine architecture as a multi-agent system. *WSCG Proceedings Part II*. Computer Science Research Notes, CSRN 2901, 27–34. https://doi.org/10.24132/CSRN.2019.2902.2.4
11. Mishra, P., Singh, U., Pandey, C., Mishra, P., & Pandey, G. (2019). Application of student's t-test, analysis of variance, and covariance. *Annals of Cardiac Anaesthesia*, 22(4), article ID 407. https://doi.org/10.4103/aca.ACA_94_19
12. Nilssen, M. G. B. (2023). Vulkan Port of OpenGL-based CUDA Snow Simulation, 131 p. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3125164
13. Popovski, F., Spasov, N., Mijakovska, S., & Nalevska, G. P. (2020). Comparison of rendering processes on 3D model. *International Journal of Computer Science & Information Technology* (*IJCSIT*), Vol. 12, 19–28. https://doi.org/10.5121/ijcsit.2020.12502
14. Sans, F., & Carmona, R. (2017). A comparison between GPU-based volume ray casting implementations: fragment shader, compute shader, OpenCL, and CUDA. *CLEI Electronic Journal*, *20*(2), 7–1. https://doi.org/10.19153/cleiej.20.2.7
15. Schütz, M., Kerbl, B., & Wimmer, M. (2021). Rendering point clouds with compute shaders and vertex order optimization. *Computer Graphics Forum*, 40(4), 115–126. https://doi.org/10.1111/cgf.14345
16. Schütz, M., Kerbl, B., & Wimmer, M. (2022). Software rasterization of 2 billion points in real time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, *5*(3), 1–17. https://doi.org/10.1145/3543863
17. Schütz, M., Mandlburger, G., Otepka, J., & Wimmer, M. (2020, May). Progressive real-time rendering of one billion points without hierarchical acceleration structures. *Computer Graphics Forum*, 39(2), 51–64. https://doi.org/10.1111/cgf.13911
18. Sultanov, A., Protsyk, M., Kuzyshyn, M., Omelkina, D., Shevchuk, V., & Farenyuk, O. (2022). A statistics-based performance testing methodology: a case study for the I/O bound tasks. *2022 IEEE 17th International Conference on Computer Sciences and Information Technologies* (*CSIT*), pp. 486–489. https://doi.org/10.1109/CSIT56902.2022.10000626
19. Vinutha, H. P., Poornima, B., & Sagar, B. M. (2018). Detection of Outliers Using Interquartile Range Technique from Intrusion Dataset. *Information and Decision Sciences: Proceedings of the 6th international conference on ficta*, pp. 511–518. https://doi.org/10.1007/978-981-10-7563-6_53
20. Xiao, L., Nouri, S., Chapman, M., Fix, A., Lanman, D., & Kaplanyan, A. (2020). Neural supersampling for real-time rendering. *ACM Transactions on Graphics* (*TOG*), 39(4), 142–148. https://doi.org/10.1145/3386569.3392376
21. Xie, F., Mishchuk, P., & Hunt, W. (2021). Real time cluster path tracing. In *SIGGRAPH Asia 2021 Technical Communications*, 1–4. https://doi.org/10.48550/arXiv.2110.08913

М. Ю. Процик, Т. О. Коротєєва

Національний університет "Львівська політехніка", м. Львів, Україна

АНАЛІЗ ЕФЕКТИВНОСТІ РОБОТИ ПРОГРАМНИХ ЗАСОБІВ ДЛЯ 3D-РЕНДЕРИНГУ В СФЕРІ РОЗРОБЛЕННЯ ВІДЕОІГОР

Наявність великої кількості доступних програмних засобів для тривимірного рендерингу дає більше простору для сучасних розробників, але й ставить їх перед непростим вибором найкращого засобу для реалізації їхніх потреб. Проаналізовано та здійснено порівняння продуктивності популярних програмних засобів для тривимірного рендерингу під час розроблення відеоігор. Для цього було розроблено простий ігровий рушій на базі архітектури ECS (англ. *Entity-Component-System*) з можливістю гнучкої конфігурації експериментальних сценаріїв на підставі JSON-файлів. Для порівняння продуктивності, в ігро-

вий рушій було інтегровано рендеринг моделей у obj-форматі на підставі трьох інтерфейсів API (англ. *Application Programming Interface*), а саме: DirectX 11, OpenGL та Vulkan. Експерименти було проведено на двох персональних комп'ютерах з інтегрованим графічним процесором Intel HD Graphics 520 та дискретним графічним процесором Nvidia GeForce RTX 4060. Вони містили рендеринг статичних та динамічних сценаріїв з використанням трьох типів тривимірних моделей різного розміру у кількостях від 1 до 120 – на інтегрованому графічному процесорі та від 1 до 2000 – на дискретному. Для кожного з виконаних спостережень було виміряно такі метрики продуктивності, як кількість оброблених кадрів за секунду, завантаженість графічного процесора, використання пам'яті графічного процесора, використання оперативної пам'яті та інші. За допомогою статистичних тестів було виявлено та підтверджено перевагу у швидкодії інтерфейсу OpenGL за відтворення експериментальних сценаріїв на інтегрованій відеокарті, як для моделей малого розміру, так і великого. Для дискретної відеокарти за рендерингу моделей великого розміру найкращі результати продемонстрував інтерфейс Vulkan, а за рендерингу моделей малого розміру – інтерфейс DirectX. У всіх протестованих сценаріях інтерфейс DirectX показав найменші витрати оперативної пам'яті та пам'яті графічного процесора. Результати дослідження будуть корисними для розробників відеоігор та інших високопродуктивних додатків у сфері тривимірного рендерингу. Також розроблений рушій, завдяки своїй гнучкості, дає змогу у майбутньому розширити перелік досліджених засобів рендерингу.

Ключові слова: комп'ютерна графіка; ігровий рушій; рендеринг у реальному часі; вимірювання продуктивності; статичний експериментальний сценарій.