Modern Graphics APIs: Design Principles, A Use Case, and New Perspectives

Lu Ping[1,2], Sun Qi[3], Wang Chen[3], Guo Jie[3], Guo Yanwen[3], Shi Wenzhe[1,2]

(1. State Key Laboratory of Mobile Network and Mobile Multimedia Technology, Shenzhen 518055, China;
2. ZTE Corporation, Shenzhen 518057, China;
3. Nanjing University, Nanjing 210093, China)

DOI: 10.12142/ZTECOM.202601013

https://kns.cnki.net/kcms/detail/34.1294.TN.20260228.1556.004.html, published online February 28, 2026

Manuscript received: 2024-06-26

Abstract: In this paper, we provide a comprehensive examination of the evolution of graphics Application Programming Interfaces (APIs). We begin by exploring traditional graphics APIs, elucidating their distinct features and inherent challenges. This sets the stage for a detailed exploration of modern graphics APIs, with a focus on four critical design principles. These principles are further analyzed through specific case studies and categorical examinations. The paper then introduces MoerEngine, a bespoke rendering engine, as a practical case to demonstrate the real-world application of these modern principles in software engineering. In conclusion, the study offers insights into the potential future trajectory of graphics APIs, spotlighting emerging design patterns and technological innovations. It also ventures to predict the development trends and capabilities of next-generation graphics APIs.

Keywords: graphics API; rendering; design principle; MoerEngine

Citation (Format 1): Lu P, Sun Q, Wang C, et al. Modern graphics APIs: design principles, a use case, and new perspectives [J]. ZTE Communications, 2026, 24(1): 97 - 106. DOI: 10.12142/ZTECOM.202601013

Citation (Format 2): P. Lu, Q. Sun, C. Wang, et al., "Modern graphics APIs: design principles, a use case, and new perspectives," *ZTE Communications*, vol. 24, no. 1, pp. 97 - 106, Mar. 2026. doi: 10.12142/ZTECOM.202601013.

1 Introduction

Graphics Application Programming Interfaces (APIs) serve as essential toolkits in graphics rendering, offering programming instructions and functions crucial for rendering 3D scenes into images and presenting frames. They bridge software with graphics hardware, enhancing the productivity of rendering and visualization. Historically, APIs like OpenGL[1] and Direct3D[2], built on a state machine model, dominated the scene but struggled with complicated rendering tasks. In contrast, modern APIs such as Vulkan[3], Direct3D 12[4], and Metal[5], are gaining traction due to their high performance and hardware optimization[6], leading to improvements in rendering efficiency and program parallelism. This shift, particularly impactful in gaming and high-performance computing, involves many explorations into general-purpose computing architectures[7 - 8], as advancements in hardware capabilities and rendering complexity push the boundaries of graphics API design and optimization.

This paper delves into the evolution and refinement of modern graphics APIs, tracing their development from early APIs like Direct3D and OpenGL. It demonstrates the limitations of traditional APIs in terms of four aspects: implicit pipeline, complicated driver layer, runtime shader compilation, and implicit task submission. The discussion then pivots to modern graphics APIs, exploring their design based on four key principles: low-level access, abstraction, separation, and parallelism. Utilizing MoerEngine, a self-developed high-performance real-time rendering engine, the paper illustrates the application of modern API design principles in real-world projects. It presents practical examples of modern API implementation in areas such as render hardware interface (RHI), render dependency graph (RDG)[9], and shader compilation. The conclusion forecasts the future trajectory of graphics APIs, exploring potential developments in low-level functionality, cross-platform compatibility, ray tracing, augmented and mixed reality, and neural network integration.

The aim of this paper is to provide a holistic view of how modern graphics APIs are revolutionizing traditional graphics rendering approaches and to speculate on their future roles and advancements in the graphics processing landscape.

This work was supported by ZTE Industry-University-Institute Cooperation Funds under Grant No. IA20230921014.

2 Limitations of Traditional Graphics APIs

The design principles of traditional graphics APIs are compatibility and ease of use. These features facilitated the rapid spread and development of traditional graphics APIs in their early stages. However, as hardware performance gradually improves and graphics tasks become more complicated, these design principles lead to numerous efficiency-related issues. This section briefly elaborates on these limitations.

2.1 Implicit Pipeline

Traditional graphics APIs use an implicit pipeline to ensure compatibility and ease of use. They expose only the interfaces for operating the graphics pipeline to software developers, making the entire graphics pipeline transparent to the application layer. This approach was developer-friendly in the early stages, as developers did not need to manage the complex graphics pipeline and could focus solely on graphics programming, objectively promoting the popularization of graphics programming.

However, with the development of GPUs and the increasing complexity of rendering tasks, this implicit pipeline has become a performance bottleneck. Developers are unable to manage the state of the graphics pipeline directly and therefore cannot perform optimization based on its current state. Moreover, they fail to apply advanced design patterns to program architecture design.

2.2 Complicated Driver Layer

The driver layer of traditional graphics APIs is overly bulky and complex. These APIs and their pipelines, exposed to software developers through a state machine model, necessitate the handling of complex tasks at the hardware driver layer. As graphics technology evolved and graphics pipelines and algorithms became more complex, the driver layer of traditional graphics APIs grew increasingly cumbersome (Fig. 1). Specifically, traditional graphics APIs embed state verification within the driver software. During application runtime, these drivers maintain and track all states while performing various validation tasks, such as confirming API markers and ensuring resource data legality.

Runtime state verification introduces additional CPU overhead and fails to maximize the use of the GPU's parallel capabilities, thereby becoming a primary bottleneck that reduces rendering performance.

2.3 Runtime Shader Compilation

Traditional graphics APIs do not provide a profound shader pre-compilation mechanism. In most traditional graphics APIs, shaders need to be compiled at runtime, with the compilation being handled by the driver (Fig. 1). Whether translating from OpenGL Shading Language (GLSL) or High-Level Shading Language (HLSL) to machine code, the process is inefficient, leading to significant shader compilation overhead. Direct3D offers DirectX Bytecode (DXBC) as precompiled code, which improves the efficiency of translating DXBC to machine code and thus enhances the shader loading speed in Direct3D. However, this also introduces the compilation overhead of DXBC. OpenGL/ES, on the other hand, leaves shader compilation entirely to the driver and does not offer a precompilation mechanism. Even though this feature can be enabled through extensions, it cannot be universally supported across all platforms[10].

Modern graphics tasks often require a variety of shaders. On traditional APIs, both the compilation and switching of shaders incur more significant performance overhead than before, which can no longer meet the requirements for high-

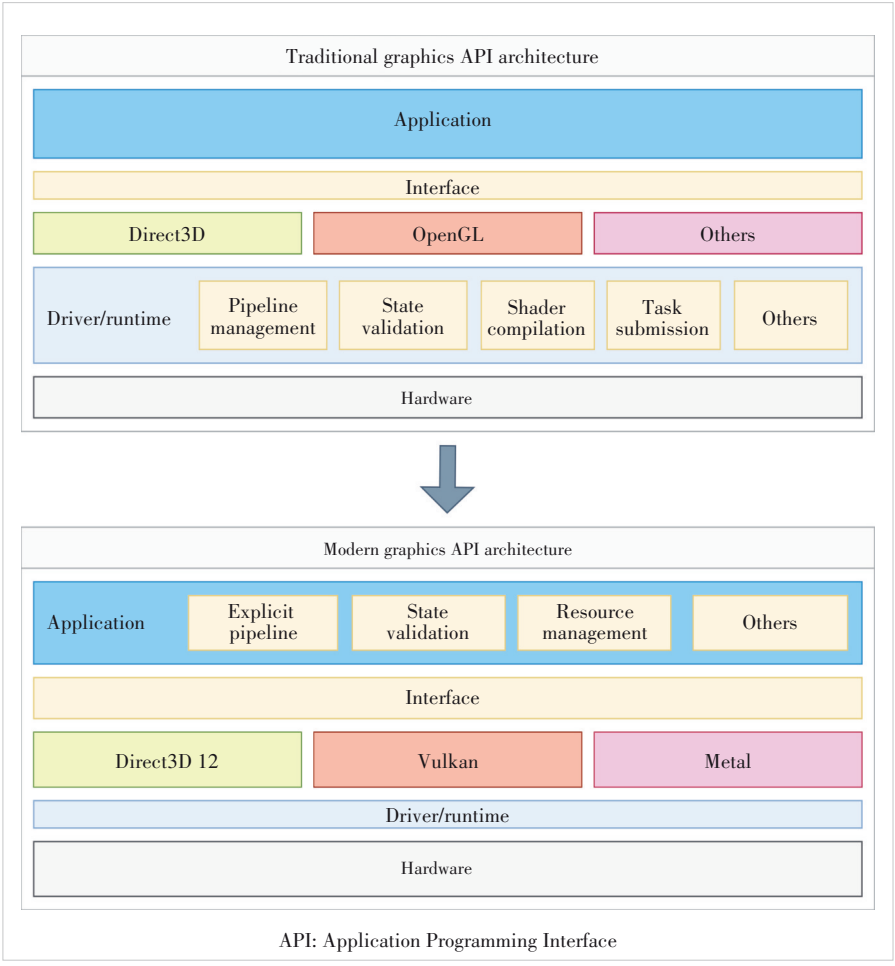

Figure 1. Comparison between traditional and modern graphics APIs

performance real-time rendering.

2.4 Implicit Task Submission

Traditional graphics APIs employ an implicit task submission model. The submission of rendering instructions is executed through an API via an application-layer-transparent runtime layer or driver (Fig. 1). When the internally maintained rendering instruction queue becomes full or when a switch in the graphics pipeline occurs, the internal runtime system or driver automatically submits the current rendering work and waits for the GPU to complete it. This significantly disrupts GPU parallelism, leading to noticeable performance degradation. Moreover, such implicit context management makes it difficult for applications to optimize synchronization between the CPU and GPU.

As hardware evolves and new rendering technologies emerge, traditional APIs are also evolving. However, in the pursuit of maintaining compatibility and ease of use, their interfaces are becoming increasingly complicated. In traditional graphics APIs, the driver manages most tasks, leading to vastly different implementations among various hardware manufacturers. The difficulty in achieving universality with different extensions is growing, as a single functionality might have multiple implementations. This results in an increasingly thick driver layer, making driver maintenance more challenging and the use of traditional APIs more complex for developers. These issues have compelled the industry to develop a new set of graphics APIs suited for modern graphics hardware and capable of meeting diverse rendering requirements, namely the modern graphics APIs.

3 Design Principles of Modern Graphics APIs

Modern graphics APIs feature a more explicit graphics pipeline, resource management strategy and synchronization mechanisms. They possess an extremely lightweight driver layer (Fig. 1), and the API itself is very close to the hardware interface, granting developers significant freedom. This enables more diverse and efficient coding design patterns. The various characteristics of modern graphics APIs have led to a leap in rendering performance.

We summarize the design principles of modern graphics APIs from various perspectives.

3.1 Low-Level Design

The most direct and crucial design principle of modern graphics APIs is their low-level approach. The design of these APIs closely mirrors hardware behavior, transferring a substantial amount of the work traditionally done at the driver layer to the application layer. Consequently, it becomes the responsibility of the application layer to manage most tasks of the modern graphics API (Fig. 1).

3.1.1 Explicit Pipeline

Modern graphics APIs utilize pipeline state objects (PSOs) to explicitly describe and manage the graphics pipeline. Aspects such as resource requirements, usage scenarios, and synchronization are all explicitly defined at the application layer, without the need for runtime-layer or driver involvement. Developers gain full control for performance optimization. If the shift from fixed to programmable pipelines marked the first revolution in graphics, the transition from implicit to explicit pipelines represents the second. Explicit pipelines offer a much larger space for performance optimization.

3.1.2 State Verification

Modern graphics APIs delegate state verification from the driver layer to the application layer. The application layer predefines the usage and format of resources, ensuring that the pipeline synchronizes properly before using resources and promptly releases resources that are no longer needed. These practices are beneficial for the driver layer and hardware to optimize rendering tasks.

3.1.3 Resource Management

Modern graphics APIs employ explicit video memory (VRAM) management. For instance, the Direct3D 12 API provides different types of heaps, allowing the application layer to allocate or release resources to these heaps. The Vulkan API defines resource types and usages, with the application layer determining the required space size and explicitly allocating or releasing various data types. The Metal API is similar to the Direct3D 12 API, where the operations at the application layer are explicit and performed with appropriately granular control.

3.2 Abstraction

Another major design principle of modern graphics APIs is abstraction. These APIs abstract resources and operations that are close to hardware, providing the application layer with a manageable space. This approach ensures the efficiency of the underlying layer while offering a more intuitive usage method. Specifically, abstraction mainly involves two aspects: resources and pipelines.

3.2.1 Resource Abstraction

Modern graphics APIs abstract resource types, storage, and access. Specifically, they classify resources based on access methods and other criteria. In terms of resource storage and access, modern APIs actively adopt an indirect addressing model. They use logical structures, as illustrated in Fig. 2, to describe the arrangement of physical data and represent resources through abstract descriptors, such as index tables. This approach helps to streamline the management and utilization of resources in graphics applications.

For instance, the Direct3D 12 API uses different heaps to store resources and employs Views and Descriptors for resource access. It also provides Root Signatures and Descriptor

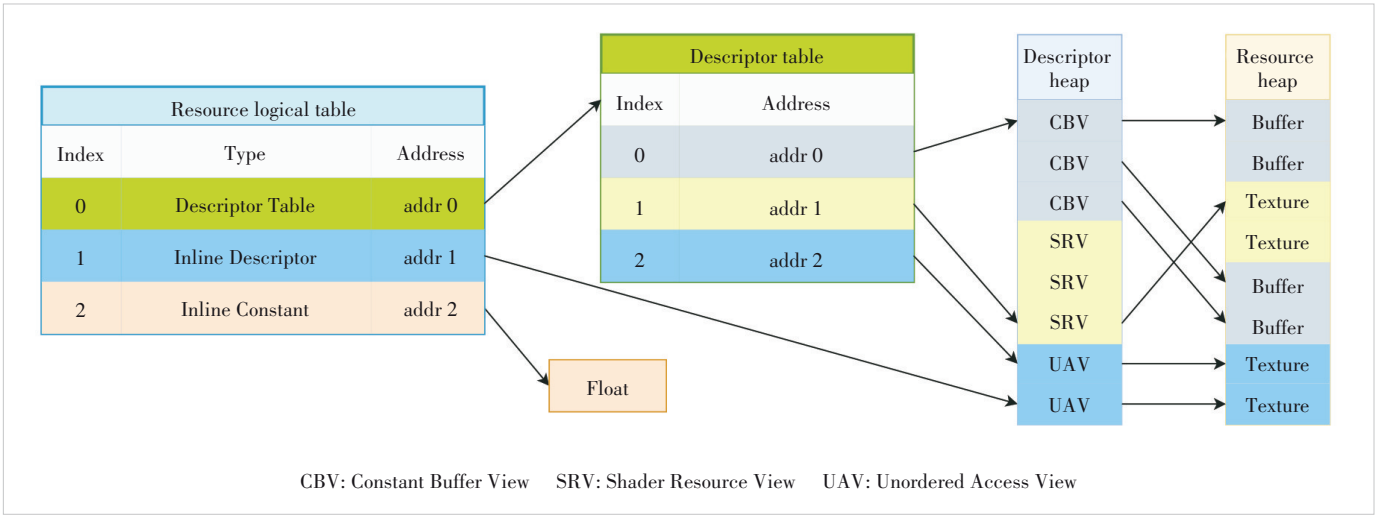

Figure 2. An indirect addressing example of modern graphics APIs

Tables to organize resources. The Vulkan API similarly abstracts resource types and access, using Pipeline Layouts and Descriptor Sets to organize resources. While the Metal API does not abstract resource access, it still follows the design principle of abstraction, organizing resources as efficiently as possible.

The resource abstraction in modern graphics APIs not only provides a flexible resource binding mechanism but also offers performance gains. For example, when switching between different pipelines, only the logical structure of the resources needs to be changed, allowing resources to be quickly adapted.

3.2.2 Pipeline Abstraction

Modern graphics APIs abstract the graphics pipeline. The pipeline is explicitly present and assembled from compiled shader modules, shader resources, and pipeline states (Fig. 3). The pipeline is created before runtime, allowing for preverification of the GPU state to ensure correctness, thereby eliminating the need for runtime checks by the driver or runtime system. The major advantage of this pipeline abstraction is speed, as it avoids runtime state checks and allows for very rapid pipeline switching.

3.3 Separation

A crucial design principle of modern graphics APIs is the separation of construction and execution. Pipeline construction, shader compilation, and resource creation are all completed prior to runtime. During runtime, only instructions are processed without touching the data. This separation enhances efficiency and performance, as it reduces the workload during runtime, allowing the GPU to focus solely on rendering tasks.

3.3.1 Pipeline Separation

The pipeline isolation in modern graphics APIs is reflected in the immutability of the pipeline once it is created. After a pipeline is fully constructed, it cannot be modified; any changes to the pipeline's properties necessitate the creation of a new pipeline. This immutability is advantageous for the runtime system, as it can fully trust the legality of the pipeline and execute it directly without the need for state checks. However, pipelines expose certain easily changeable parameters.

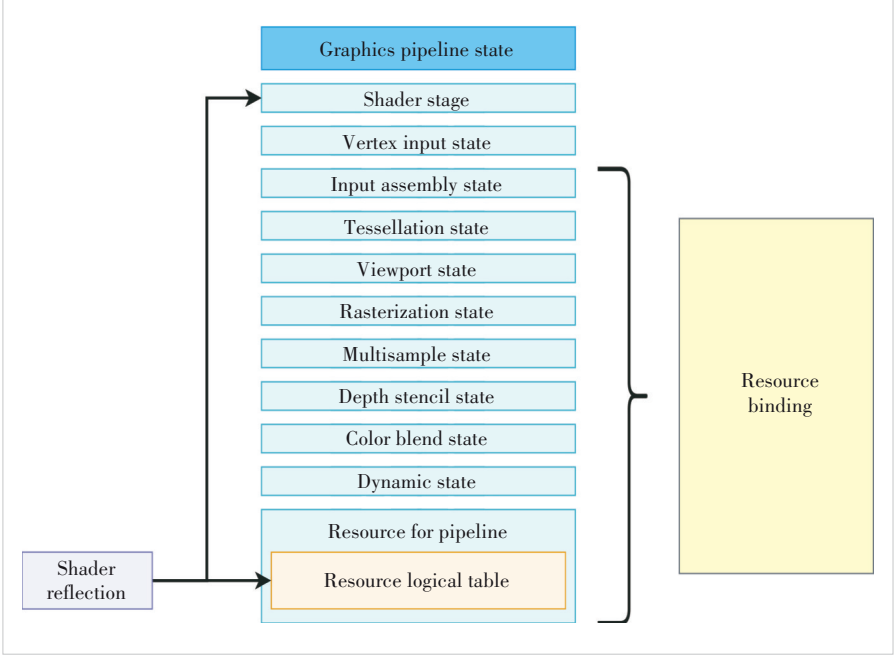

Figure 3. Pipeline layout of modern graphics APIs

For example, in the Vulkan API, the graphics pipeline exposes properties like the viewport and scissor. Similarly, in the Direct3D 12 API and Metal API, various types of pipelines also expose certain properties.

3.3.2 Shader Separation

Shader isolation in modern graphics APIs is exemplified by the shader pre-compilation mechanism and pipeline assembly mechanism. Modern graphics APIs provide shader compilers for precompiling shaders into bytecode, which offers significant performance advantages when loading and assembling pipelines. Additionally, the runtime system does not need to concern itself with shader loading and management, as these tasks are already completed during pipeline assembly. Common shader compilers, such as GLSLang Compiler and DirectX Shader Compiler, offer comprehensive support for various shading languages including GLSL, HLSL, and Slang[11]. This approach streamlines the integration of shaders into the graphics pipeline and optimizes their performance during rendering tasks.

3.3.3 Resource Separation

Resource isolation in modern graphics APIs is manifested in the mechanisms for updating and binding resources. Resources required at runtime are created beforehand and submitted to the pipeline through binding. These APIs provide explicit update mechanisms. When a program is running and using a bound resource, updating that resource is not allowed, ensuring API-level resource correctness. The initial version of the Vulkan API even prohibited updating resources after binding. Direct3D 12, owing to its resource organization, supports updating resources after binding while enforcing validity constraints. Metal also exhibits resource isolation characteristics, with its implementation being more hardware-centric and requiring fewer complex operations at the application layer. This approach ensures stability and efficiency in resource management during the rendering process.

3.4 Parallelism

Modern graphics APIs also focus on parallelism, a feature that significantly distinguishes them from traditional graphics APIs and allows for the full utilization of the powerful capabilities of both CPUs and GPUs. There are three types of parallelism: CPU to GPU parallelism, intra-GPU parallelism, and inter-GPU parallelism.

Modern graphics APIs' emphasis on these parallelism types results in substantial performance improvements[12], especially in applications that require intense graphics processing, such as modern video games and professional graphics design software.

3.4.1 Parallelism Between CPU and GPU

When the CPU calls a modern graphics API, the GPU does not immediately execute the corresponding instructions. Instead, the commands are saved to a list, which is only executed after being submitted to the GPU. Typically, each thread holds a command list, and different threads can record commands simultaneously and submit them in parallel to the GPU (Fig. 4). This process avoids prolonged idle times for the CPU, thereby improving CPU utilization.

3.4.2 Parallelism in GPU

Modern graphics APIs are capable of accessing different engines on the GPU, such as the 3D Engine, Compute Engine, and Copy Engine (Fig. 5). These engines can operate in parallel at the hardware level and are more efficient in handling their respective tasks. The CPU submits tasks to different types of queues, which are eventually dispatched to the corresponding engines. Submitting tasks according to their types significantly enhances the degree of concurrency within the GPU.

3.4.3 Parallelism Between GPUs

Modern graphics APIs explicitly support parallelism across multiple GPUs (Fig. 6). Different resources can be allocated to different GPUs, and different commands can be submitted to separate GPUs. After execution, the results from multiple GPUs are consolidated onto a single GPU used for displaying the results. This inter-GPU parallelism offers significant performance improvements when dealing with large-scale scenes and extensive tasks.

4 A Use Case: MoerEngine

MoerEngine is a high-performance real-time rendering engine developed using modern graphics APIs. Its core render module consists of the RHI, the RDG, and the shader compiler.

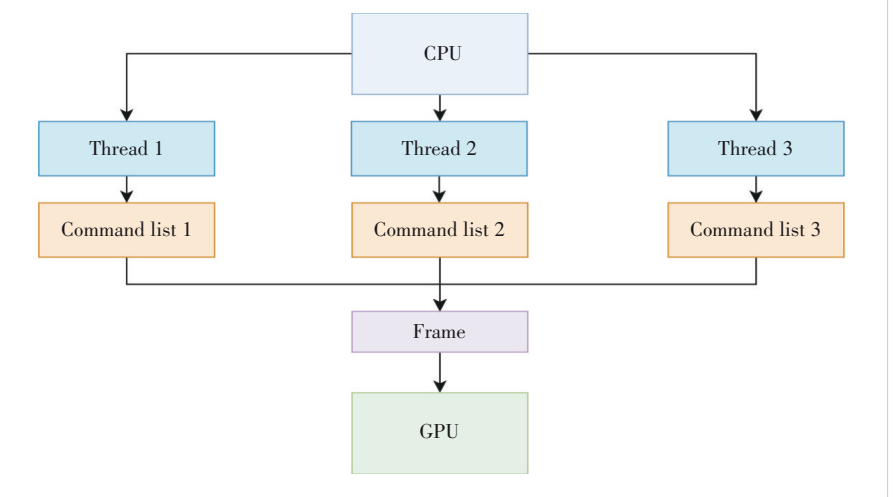

Figure 4. CPU multithreading in rendering

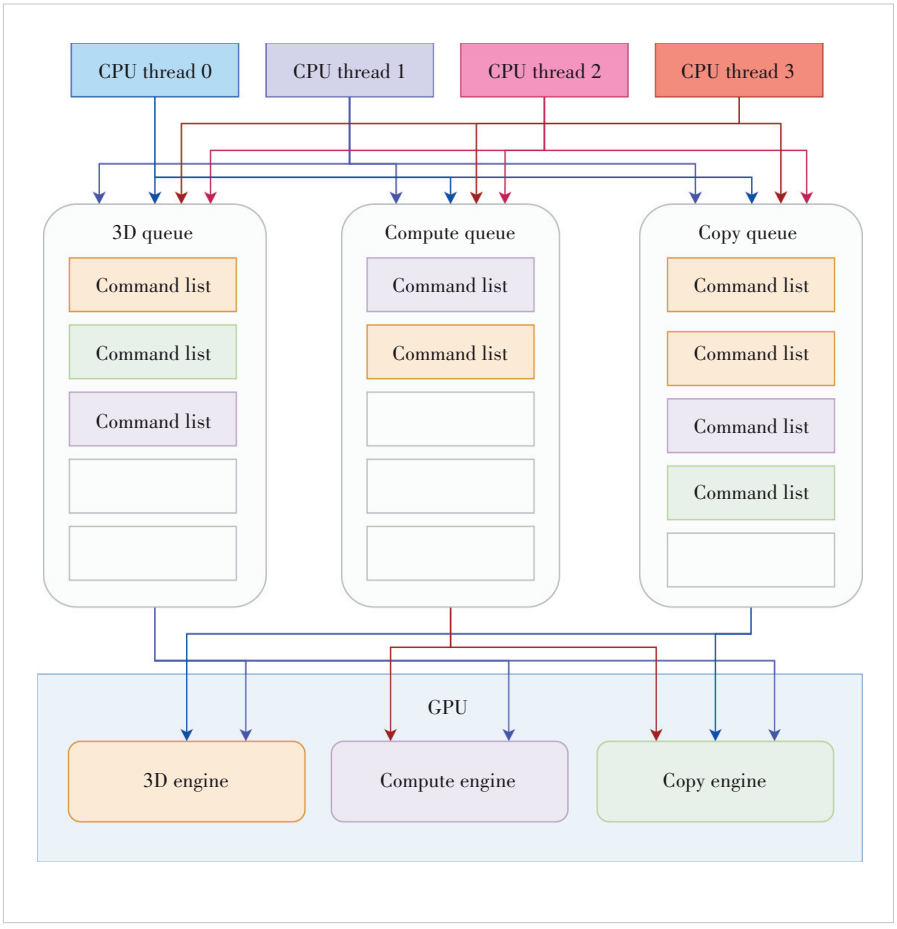

Figure 5. GPU internal parallelism

In this section, we elaborate on the development principles of modern graphics APIs in the context of MoerEngine's render modules.

In the render module of MoerEngine, a single component may reflect multiple design principles of modern graphics APIs (Fig. 7). This is quite common in modern engine development, where good design patterns often comprehensively consider the design principles of modern graphics APIs.

4.1 Challenge

Through the explanations of traditional and modern graphics APIs, it is evident that modern graphics APIs are explicitly designed, approaching the functionality of the driver layer, and require developers to undertake some of the tasks traditionally managed by the driver. This design grants developers greater freedom and, in theory, has the potential to maximize real-time rendering performance. However, achieving this is not straightforward; it requires more management and development work within the rendering engine. Simply porting an existing engine to a modern graphics API without thorough integration may result in performance that is inferior to that of traditional graphics APIs.

This section, based on the self-developed rendering engine MoerEngine, discusses the utilization of modern graphics APIs in high-performance rendering engines. MoerEngine supports multiple graphics APIs, whose designs are largely similar but differ in details. Therefore, the choice of a specific API is not the focus of this article; instead, we concentrate primarily on design approaches rather than specific API code.

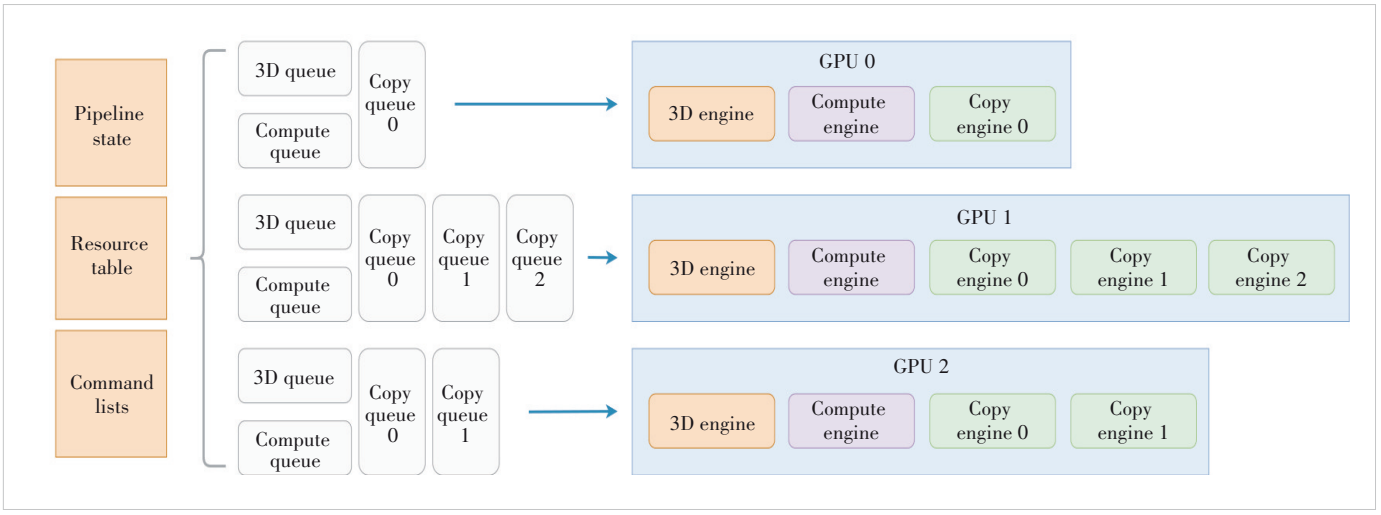

Figure 6. Parallelism across multiple GPUs

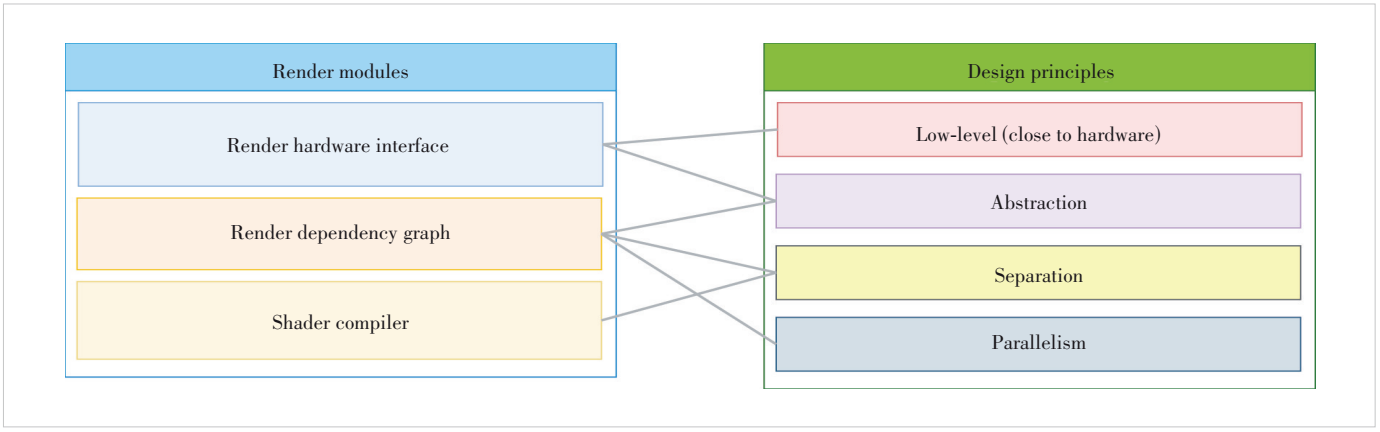

Figure 7. Modern API design principles in MoerEngine render modules

4.2 Render Hardware Interface

Render hardware interface (RHI) is an abstraction layer in the rendering engine for multiple platform-specific graphics APIs. It is designed from the ground up to fully exploit the advantages of various modern graphics APIs across different platforms. RHI embodies the low-level and abstraction principles of modern graphics APIs. The ability of MoerEngine to support multiple graphics APIs is largely attributed to the RHI.

RHI takes on the task of encapsulating low-level modern graphics APIs. It standardizes more generic implementations found in graphics APIs while distinguishing specialized implementations through different interfaces. RHI also conceals the complexities of the original APIs, such as intricate parameters, buffering, and multi-threaded scheduling, allowing higher-level modules to focus solely on the interfaces provided by RHI. Furthermore, RHI can internally perform performance optimizations or platform adaptations without changing its external interfaces. This capability of RHI enables a more efficient and adaptable interaction with the underlying graphics APIs, significantly simplifying the development process for the upper layers of the rendering engine.

4.3 Render Dependency Graph

Render dependency graph (RDG) is a higher-level abstraction within the rendering engine, representing a design pattern for managing complex rendering pipelines. It is a graph-based scheduling system designed to perform whole-frame optimization of the rendering pipeline and is widely used in the industry. RDG leverages modern graphics APIs for automatic asynchronous compute scheduling and more effective memory and barrier management to enhance performance (Fig. 8). It embodies the abstraction, separation, and parallelism design principles of modern graphics APIs. MoerEngine uses RDG to register and schedule rendering tasks for high-performance rendering.

Modern graphics APIs enable advanced rendering pipeline contexts to schedule rendering tasks, thereby improving performance and simplifying the rendering task stack. RDG delays task execution, recording the entire frame's rendering tasks into a dependency graph data structure. Once all passes are collected, the dependency graph is compiled and executed in an order sorted by dependencies. With the whole-frame context of the dependency graph and the powerful features of modern graphics APIs, RDG can perform complex scheduling tasks transparently to the user. Its specific capabilities include:

• automatically scheduling and synchronizing asynchronous compute channels;

• keeping resources' memory active during non-continuous intervals;

• pre-emptively starting barriers and layout transitions to avoid pipeline stalls.

Furthermore, RDG utilizes the dependency graph to provide extensive validation, enabling the automatic capture of functional and performance issues to improve the development process.

4.4 Shader Compiler

In modern graphics APIs, shaders generally support a pre-compilation mechanism and independent compilers, which

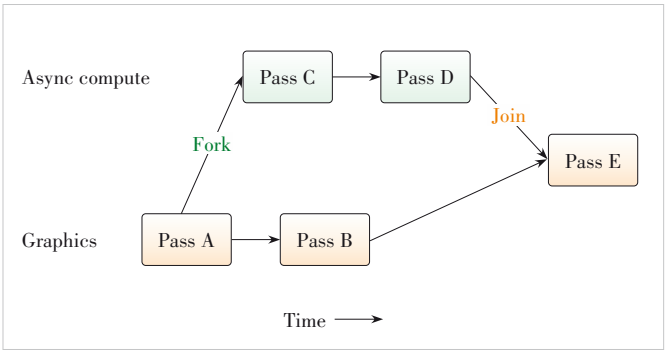

Figure 8. Render Dependency Graph in MoerEngine

has led to the widespread adoption of a general shader reflection mechanism. Shader reflection is not a new feature, as it has been supported since Direct3D 10[13], but widespread adoption has only occurred with modern graphics APIs. The shader reflection mechanism offers distinct advantages: it allows for the acquisition of resource information needed by the shader and automates resource binding to the pipeline during the pipeline construction phase. Shader compilation and reflection exemplify the separation principle of modern graphics APIs.

Specifically, various modern graphics APIs have their own shader pre-compilation formats, such as Direct3D 12's DXIL[14] and Vulkan's SPIR-V[15]. Metal differs from the others in that its intermediate representation resembles LLVM bytecode[16]. There are many shader compilers available that can compile shader languages into intermediate code forms, with reflection being carried out on the intermediate representations. MoerEngine embraces shader pre-compilation and reflection mechanisms, automatically aligning shader resources with pipeline resources. In MoerEngine, the DirectX Shader Compiler is used for shader compilation in Direct3D 12 and Vulkan, and corresponding API reflection is utilized (Fig. 9). This process significantly streamlines shader management, ensuring efficient and accurate shader resource utilization in the rendering pipeline.

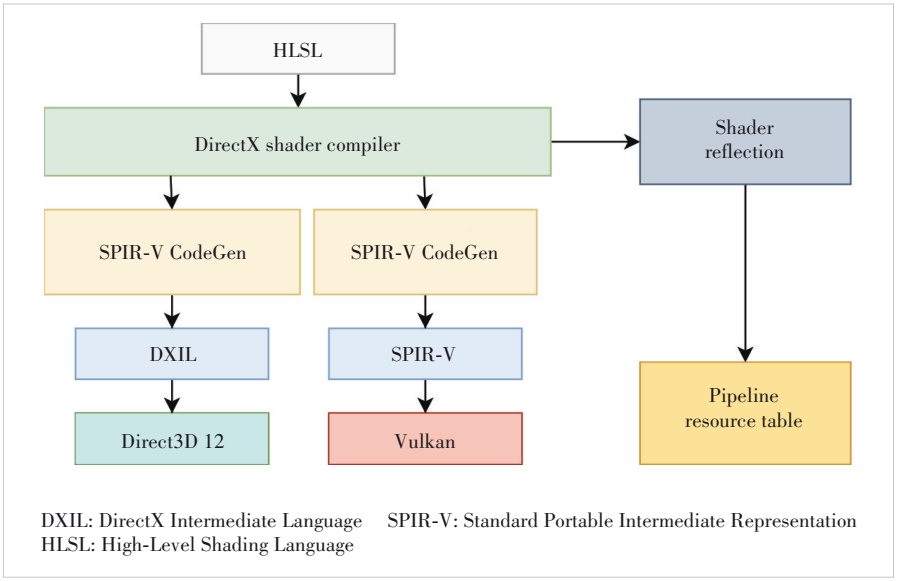

Figure 9. MoerEngine shader pre-compilation and reflection

4.5 Results

MoerEngine, developed based on the design principles of modern graphics APIs, achieves high performance and robust availability. It leverages real-time ray tracing algorithms to render complex scenes at interactive frame rates with high visual fidelity. Fig. 10 presents example scenes rendered in real time using MoerEngine. These scenes contain from hundreds of thousands to millions of triangles, featuring a variety of complex lighting and material types. All scenes are rendered using an NVIDIA 3090 GPU. With real-time ray tracing and denoising, MoerEngine achieves performance of over 90 frames per second (fps).

5 Future of Modern Graphics APIs

Modern graphics APIs are currently widely applied, meeting the rapidly growing demand for efficient graphics rendering. Especially in recent years, we have seen the emergence of new features built upon these APIs, such as Epic Games' GPU-driven Nanite and Lumen[17] technologies, which have elevated real-time rendering to new heights. Additionally, APIs like Direct ML and Vulkan ML have ventured into the field of machine learning, fully leveraging GPU's parallel computing power to accelerate various machine learning tasks. With these new features, modern graphics APIs have become more versatile, capable of handling increasingly complex and diverse tasks. This section explores the future of graphics APIs and graphics rendering based on the evolution of hardware and rendering requirements.

5.1 Evolution of Low-Level APIs

Low-level APIs such as Direct3D 12, Vulkan, and Metal are likely to continue evolving, offering closer control to the hardware layer. This would allow developers to more efficiently utilize the performance of modern multi-core GPUs, such as more direct and finer-grained control of VRAM and

Figure 10. Scenes rendered by MoerEngine

more direct thread control, potentially raising the performance ceiling of these APIs.

5.2 Cross-Platform Rendering

Modern graphics APIs have platform limitations; even Vulkan cannot achieve full hardware performance on every platform with a single codebase. Cross-platform graphics APIs, like WebGPU[18], are expected to be further refined and adopted. These APIs are not limited merely to platform compatibility; they also offer heterogeneous multi-device rendering and the potential for end-to-cloud rendering. Combined with remote calling mechanisms, cross-platform APIs could integrate rendering resources more effectively.

5.3 Native Real-Time Ray Tracing

Some APIs have already begun to support real-time ray tracing, with Microsoft's DirectX providing the DirectX Raytracing (DXR)[19] library for native ray tracing support. Mobile hardware manufacturers are now incorporating ray tracing acceleration units into their GPUs, indicating that real-time ray tracing is the future. Upcoming graphics APIs will inevitably support ray tracing more natively, possibly introducing more specialized ray tracing features or acceleration mechanisms.

5.4 Augmented and Mixed Reality Support

As virtual reality (VR), augmented reality (AR), and mixed reality (MR) technologies evolve, real-time rendering needs to support higher resolutions and faster refresh rates to provide a more immersive user experience[20]. AR and MR technologies, in particular, require rendering results that match real-world information, such as real-world lighting and color tones. Graphics APIs will need to provide access to a richer set of hardware capabilities, enabling applications to read real-world information and accurately reproduce it in virtual environments.

5.5 Neural Network Integration

In modern graphics processing, neural networks are playing an increasingly important role[21]. The trend in graphics hardware development now includes more hardware units optimized for neural network computations. Graphics APIs must also accommodate neural network tasks and integrate them more deeply with graphics pipelines, for example, through more direct input of physical information into neural networks to assist rendering. Müller et al.[22] utilized a neural radiance cache to accelerate global illumination in path tracing, significantly enhancing real-time ray tracing performance. With the advantages of modern hardware capabilities, the overhead for cache updates and queries can be extremely low. Going further, graphics APIs could completely restructure graphics pipelines, relying solely on neural networks for rendering. Such APIs would have a completely different design principle from traditional graphics pipelines, focusing on how to maximize neural network performance and provide the networks with the flexibility to handle diverse scenarios.

6 Conclusions

In conclusion, this paper has thoroughly examined the development and impact of modern graphics APIs such as Direct3D 12, Vulkan, and Metal, highlighting their revolutionary role in enhancing rendering efficiency. It underscores the significant advancements these APIs have brought to the field of graphics rendering, facilitating the transition from traditional to more sophisticated, high-performance techniques. Looking forward, this paper anticipates further innovations in graphics technology that are poised to reshape graphical computing in various applications. This study not only contributes to our understanding of current API capabilities but also paves the way for future research in evolving graphics technologies.

References

[1] Khronos Group, Inc. OpenGL registry [EB/OL]. (1992-06-30) [2022-05-05]. https://registry.khronos.org/OpenGL/index_gl.php
[2] Microsoft. Direct3D [EB/OL]. (1996-06-02) [2021-09-11]. https://learn.microsoft.com/en-us/windows/win32/direct3d
[3] Khronos Group, Inc. Vulkan specification [EB/OL]. (2016-02-16) [2023-12-08]. https://registry.khronos.org/vulkan/specs
[4] Microsoft. Direct3D 12 programming guide [EB/OL]. (2015-07-29) [2021-12-30]. https://learn. microsoft. com/en-us/windows/win32/direct3d12/directx-12-programming-guide
[5] Apple Inc. Metal [EB/OL]. [2023-12-08]. https://developer. apple. com/documentation/metal
[6] Unterguggenberger J, Kerbl B, Wimmer M. Vulkan all the way: transitioning to a modern low-level graphics API in academia [J]. Computers & graphics, 2023, 111: 155 – 165. DOI: 10.1016/j.cag.2023.02.001
[7] Thompson C J, Hahn S, Oskin M. Using modern graphics architectures for general-purpose computing: a framework and analysis [C]//Proc. 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35. IEEE, 2002: 306 – 317. DOI: 10.1109/MICRO.2002.1176259
[8] Owens J D, Luebke D, Govindaraju N, et al. A survey of general-purpose computation on graphics hardware [J]. Computer graphics forum, 2007, 26 (1): 80 – 113. DOI: 10.1111/j.1467-8659.2007.01012.x
[9] O'Donnell Y. (GDC 2017) FrameGraph: extensible rendering architecture in frostbite [R]. San Francisco, CA: Frostbite/Electronic Arts, 2017
[10] Piñeiro A. ARB_gl_spirv: bringing SPIR-V to Mesa OpenGL: FOSDEM 2018 [R]. Brussels: Igalia, 2018
[11] Bangaru S P, Wu L F, Li T-M, et al. 2023. SLANG.D: fast, modular and differentiable shader programming [J]. ACM trasactions on graphics, 2023, 42(6): 1 – 28. DOI: 10.1145/3618353
[12] John McDonald. (GTC 2016) High performance vulkan: lessons learned from source 2 [R]. San Jose: Valve, 2016
[13] Blythe D. The Direct3D 10 system [J]. ACM transactions on graphics, 2006, 25(3): 724 – 734. DOI: 10.1145/1141911.1141947
[14] Microsoft. DirectX intermediate language [EB/OL]. (2016-12-29) [2023-12-01]. https://github. com/microsoft/DirectXShaderCompiler/blob/main/docs/DXIL.rst
[15] Khronos Group, Inc. SPIR-V specification [EB/OL]. (2015-05-03) [2021-12-16]. https://www.khronos.org/registry/SPIR-V
[16] Aras Pranckevičius. Porting unity to new APIs [C]//Proc. ACM SIGGRAPH 2015 Course Notes: An Overview of Next-generation Graphics APIs. DOI: 10.1145/2776880.2787704
[17] Tatarchuk N, Dupuy J, Deliot T, et al. Advances in real-time rendering in

games: part I [C]//Proc. ACM SIGGRAPH 2022 Courses. ACM, 2022. DOI: 10.1145/3532720.3546895
[18] Kenwright B. Introduction to the WebGPU API [C]//Proc. ACM SIGGRAPH 2022 Courses. ACM, 2022. DOI: 10.1145/3532720.3535625
[19] Matt Sandy. (GDC 2018) DirectX Raytracing [R]. San Francisco: Microsoft, 2018
[20] Billinghurst M, Nebeling M. Rapid prototyping of XR experiences [C]// Proc. ACM SIGGRAPH 2022 Courses. ACM, 2022. DOI: 10.1145/3532720.3535684
[21] Marshall C S. Practical machine learning for rendering: from research to deployment [C]//Proc. ACM SIGGRAPH 2021 Courses. ACM, 2021. DOI: 10.1145/3450508.3464564
[22] Müller T, Rousselle F, Novák J, et al. Real-time neural radiance caching for path tracing [J]. ACM transactions on graphics, 2021, 40(4): 1 – 16. DOI: 10.1145/3450626.3459812

Biographies

Lu Ping is the Vice President and Director of the R&D Project in the Technology Planning Department at ZTE Corporation. He also serves as Executive Deputy Director of the National Key Laboratory of Mobile Network and Mobile Multimedia Technology. His research directions include cloud computing, big data, augmented reality, and multimedia service-based technologies. He has contributed to major national science and technology projects and has published multiple papers and authored two books.

Sun Qi (522023330087@smail.nju.edu.cn) is a master's student in the Department of Computer Science and Technology at Nanjing University, China, affiliated with the Meta Graphics & 3D Vision Lab. His research interests include photorealistic rendering and high-performance real-time rendering.

Wang Chen is a master's student in the Department of Computer Science and Technology at Nanjing University, China, affiliated with the Meta Graphics & 3D Vision Lab. His research interests include real-time ray tracing and rendering software architecture.

Guo Jie is an associate researcher at the Department of Computer Science and Technology, Nanjing University, China. He received his PhD from Nanjing University in 2013. His current research interests include computer graphics, virtual reality, and 3D vision. He has authored over 80 publications in leading international conferences (e.g., SIGGRAPH, SIGGRAPH Asia, CVPR, ICCV, ECCV, IEEE VR) and journals (e.g., *ACM ToG*, *IEEE TVCG*, *IEEE TIP*). He has developed several applications in illumination prediction, material prediction, and real-time rendering, which have been widely used in industry and achieved good economic and social benefits.

Guo Yanwen is a Professor and PhD supervisor at Nanjing University, China. He was appointed as a PhD supervisor in July 2013 and as a Professor in December 2014. He is a recipient of the Jiangsu Province Outstanding Young Scientist Fund and a core member of the Department of Computer Science and Technology and the State Key Laboratory for Novel Software Technology at Nanjing University. He serves as Executive Director of the Nanjing University-iQIYI Joint Innovation Center, a board member of the China Society for Image and Graphics, Chair of the Graphics and Image Committee of the Jiangsu Computer Society, and Chair of the Virtual Reality Committee of the Jiangsu Society of Engineers. He has published nearly 100 high-level papers, including approximately 20 in ACM/IEEE Transactions.

Shi Wenzhe is a strategy planning engineer at Beijing Xingyun Digital Technology Co., Ltd. and a member of the National Key Laboratory for Mobile Network and Mobile Multimedia Technology, China. He is also involved inproduct planning for the XRExplore Platform at Beijing Xingyun Digital Technology Co., Ltd. His research interests include indoor visual AR navigation, SFM 3D reconstruction, visual SLAM, real-time cloud rendering, VR, and spatial perception.

New Member of ZTE Communications Editorial Board

Yu Zhiwen is the Vice President of Harbin Engineering University, China, and a professor at Northwestern Polytechnical University, China. He is a Fellow of the China Computer Federation (CCF), a Distinguished Professor of the Chang Jiang Scholars Program of the Ministry of Education, China, a recipient of the National Science Fund for Distinguished Young Scholars, China, a leader in science and technology innovation under the National Ten Thousand Talents Program, China, and the Chief Scientist of a National Key R&D Program Project, China.

He serves as the Director of the Ministry of Education Key Laboratory of Human-Machine-Object Fusion and Crowd Intelligence Computing, China, the Director of the Ministry of Industry and Information Technology Key Laboratory of Intelligent Perception and Computing, China, and the Leader of the National Innovation Team for Underwater Swarm Intelligence, China.

His research interests include the Internet of Things (IoT), ubiquitous computing, and crowd intelligence sensing and computing. He has published over 300 papers in top international academic journals and conferences, including *IEEE TMC*, *IEEE TKDE*, MobiCom, UbiComp, INFOCOM, and KDD, with 9 papers recognized as ESI Highly Cited Papers.

He is an editorial board member of prestigious international journals such as *IEEE Transactions on Human-Machine Systems*, *IEEE Communications Magazine*, and *ACM IMWUT*. He has also served as conference chair or program committee member for over 50 international conferences, including ACM UbiComp, IEEE PerCom, and IJCAI.

Additionally, he holds positions such as Chair of the ACM Xi'an Chapter, China, Senior Member of IEEE, Executive Council Member of the CCF, and Chair of the Technical Committee on Ubiquitous Computing of the CCF.

He has received numerous awards, including the IEEE HITC Outstanding Technical Achievement Award, the IEEE Smart World Outstanding Research Award, the CCF Excellent Doctoral Dissertation Award, the CCF Young Scientist Award, the Second Prize of National Teaching Achievement, the First Prize of Natural Science of the Ministry of Education, China, the First Prize of Natural Science of Shaanxi Province, and the First Prize of Science and Technology Progress of Heilongjiang Province.