A comparative analysis of graphics APIs and shader languages used in modern game engines and their impact on the game development process

Emil Stephansen

Abstract This paper presents a comparative analysis of graphics **Application Programming Interface (API)**s, shader languages, and shader development workflows in modern game engines, with a specific focus on Unity, Unreal Engine, and Godot. Using data extracted from the **International Game Database (IGDB)**, the study investigates trends in engine adoption, **Graphics Processing Unit (GPU) API** usage, and shader tools employed across multiple platforms.

The analysis highlights that Unity and Unreal Engine dominate the market, with **High-Level Shader Language (HLSL)** emerging as the most widely used shader language due to its strong integration with DirectX and Vulkan. Metal Shading Language (**Metal Shading Language (MSL)**) and **PlayStation Shader Language (PSSL)** remain relevant in platform-specific ecosystems like Apple and PlayStation. Furthermore, the study reveals significant differences in shader workflows:

- Unreal Engine excels in high-fidelity rendering with its robust Material Editor but poses challenges for custom **HLSL** shaders.
- Unity's Shader Graph provides accessibility and flexibility, catering to both **Universal Render Pipeline (URP)** and **High Definition Render Pipeline (HDRP)**.
- Godot offers a streamlined custom shader language and visual tools optimized for lightweight workflows.

The study also discusses the ongoing deprecation of OpenGL in favor of modern **API**s like Vulkan and Metal, driven by the need for improved performance and visual fidelity. By analyzing engine-specific rendering systems, we provide insights into the trade-offs between flexibility, accessibility, and platform optimization.

This work aims to guide game developers in selecting appropriate tools and workflows for shader programming, balancing visual quality, performance, and development efficiency.

doi:N/A

Introduction

The rapid advancement of computer graphics has significantly influenced the game development process, enabling developers to produce visually stunning and complex interactive experiences. Central to this progress are the underlying graphics **API**s and shader languages that dictate how visual data is processed and rendered on modern hardware. Choosing the appropriate tools, such as game engines, rendering pipelines, and shader workflows, is essential for balancing visual fidelity, performance, and development efficiency.

Two engines, Unity and Unreal Engine, have emerged as dominant platforms in the game development industry. Unity's accessibility, cross-platform versatility, and customization options make it particularly attractive for small teams and mobile development. Meanwhile, Unreal Engine prioritizes photorealism and advanced rendering capabilities, positioning it as the preferred choice for high-end projects. The rise of open-source alternatives like Godot further expands developers' options, particularly for lightweight and flexible workflows.

Shader programming plays a crucial role in modern rendering pipelines, driving visual effects such as lighting, shadows, reflections, and post-processing. The choice of shader language—whether **HLSL**, **OpenGL Shading Language (GLSL)**, or platform-specific options like **MSL**—directly influences the complexity of development and the achievable graphical quality. Simultaneously, the industry-wide shift from legacy **API**s like OpenGL to modern alternatives like Vulkan, DirectX 12, and Metal underscores the demand for optimized performance on diverse hardware platforms.

Despite the wealth of tools available, there remains a lack of detailed analyses comparing shader development workflows across engines. While existing research explores engine performance and high-level features, few studies dive into the practical implications of shader programming tools, **API** selection, and their impact on development workflows.

Purpose and Contribution

This paper addresses this gap by conducting a comparative analysis of:

1. Graphics **API**s used across platforms (DirectX, Vulkan, Metal).
2. Shader languages and their adoption in Unity, Unreal Engine, and Godot.
3. Shader workflows—visual tools (e.g., Shader Graph, Material Editor) versus code-based programming.

The study leverages data from the **IGDB** to identify trends in engine usage and **API** adoption. By analyzing the trade-offs between flexibility, performance, and accessibility, this work aims to guide developers in selecting tools and workflows suited to their project goals, whether targeting high-fidelity visuals, lightweight solutions, or rapid prototyping.

Related Work

Recent research on game engines and their underlying technologies has focused on several key aspects that are crucial for understanding how these systems impact game development: graphics **API**s, shader languages, performance, workflow efficiency, and the balance between visual quality and resource usage. In this section, we provide an overview of the most significant findings in these areas, which helps to illustrate the trade-offs and strengths of different engines, particularly Unity and Unreal Engine, as well as emerging alternatives like Flax.

One major focus in recent years has been on the move towards physically-based shading models. Karis[1] explored the implementation of such a model in Unreal Engine 4, aiming to enhance visual realism while simplifying parameter complexity. This work laid the foundation for a modern shading workflow that balances visual quality with computational efficiency. This trend towards physically-based shading is evident across many engines, as developers strive for photorealism without sacrificing usability.

Another important dimension of comparison involves the practical use cases of different engines. Sabir et al.[2] investigated the use of Unity and Unreal Engine for synthetic data generation in construction hazard scenarios. Their findings showed that while Unreal Engine's advanced graphical capabilities made it ideal for simulating visually complex environments, Unity's accessibility and ease of use positioned it as a more suitable choice for rapid prototyping. This highlights a recurring theme in engine selection: the trade-off between high-end visual fidelity and development speed.

The versatility of Unity has also been highlighted in broader contexts. Hussain et al.[3] conducted a technical survey of Unity, emphasizing its cross-platform capabilities and growing use beyond traditional gaming, including in VR/AR and other industries. Unity's simplicity and versatility make it attractive to both beginners and professionals, particularly those looking to develop across a range of platforms without needing specialized expertise.

In terms of engine architecture, Lee et al.[4] compared Unity and Unreal Engine 4 by implementing a first-person shooter in both. Their study found that Unity's component-based system offers greater flexibility for smaller teams, while Unreal's tools are better suited for achieving photorealistic visuals. This comparison points to a fundamental difference in how these engines approach development: Unity prioritizes modularity and ease of use, whereas Unreal Engine provides robust tools for detailed, high-quality graphics.

1. Karis, Brian, Real Shading in Unreal Engine 4.

2. Sabir, Aqsa, et al., Synthetic Data Generation with Unity 3D and Unreal Engine for Construction Hazard Scenarios: A Comparative Analysis.

3. Hussain, Faizan, et al., Unity Game Development Engine: A Technical Survey.

4. Lee, HanSeong, SeungTaek Ryoo, and SangHyun Seo, A Comparative Study on the Structure and Implementation of Unity and Unreal Engine 4.

Performance analysis has also been a major focus of comparative studies. Szelug[5] compared Unity with the newer Flax Engine, examining their physics simulations and general performance. Unity consistently outperformed Flax in terms of frame rate and resource efficiency, which is perhaps expected given Flax's relatively recent entry into the market and ongoing development. Nevertheless, this comparison helps highlight the challenges and opportunities for emerging engines attempting to compete with established industry leaders.

A deeper look at performance in 3D games was provided by Abramowicz and Borczuk[6], who found that Unity tends to achieve a better average frame rate while using fewer resources compared to Unreal Engine. However, Unreal Engine was noted for providing superior visual quality at the cost of higher RAM and **GPU** usage. This trade-off between performance and graphical fidelity is a recurring theme in engine selection, depending on project goals and available hardware.

Szabat and Plechawska-Wójcik[7] added another layer to this discussion by comparing user experiences with Unity and Unreal. They found that while Unity generally provides better runtime performance, Unreal's visual quality tends to receive higher user satisfaction ratings. This suggests that the choice of engine may depend not only on technical performance metrics but also on the desired end-user experience.

Šmíd[8] approached the comparison through a practical benchmark involving a reimplementation of the classic Pac-Man game. His work emphasized Unity's suitability for prototyping due to its simplicity, while Unreal's sophisticated features allowed for deeper customization at the cost of a steeper learning curve. This illustrates how different engines cater to varying levels of developer expertise and project complexity.

Finally, Szafran and Plechawska-Wójcik[9] examined the impact of graphics settings on performance, specifically in Unreal Engine. Their results confirmed that changes in shadow quality had the most significant effect on frame rates, emphasizing the importance of careful graphics tuning to balance visual quality and performance.

Together, these studies provide a comprehensive overview of the strengths and weaknesses of modern game engines. Understanding these trade-offs allows developers to make informed choices about which graphics **API**s and shader languages are most suitable for their projects, whether they aim for cross-platform versatility, high-end visual fidelity, or rapid prototyping capabilities.

5. Szelug, Wojciech, Comparative Analysis of the Performance of the Flax Engine and Unity.

6. Abramowicz, Kamil, and Przemysław Borczuk, Comparative Analysis of the Performance of Unity and Unreal Engine in 3D Games.

7. Szabat, Bartłomiej, and Małgorzata Plechawska-Wójcik, Comparative Analysis of Selected Game Engines.

8. Šmíd, Antonín, Comparison of Unity and Unreal Engine.

9. Szafran, Kamil, and Małgorzata Plechawska-Wójcik, Impact of Changes in Graphics Setting on Performance in Selected Video Games.

Gaps in Existing Research

While the existing body of research provides valuable insights into the performance and usability of different game engines, there are several notable gaps. Most studies focus on the general development workflows and performance of Unity and Unreal Engine, without diving deeply into the specific nuances of shader development itself. The emphasis tends to be on the ease of use and high-level graphical features rather than the intricacies of shader programming and customization.

Furthermore, almost all comparative analyses are limited to the two major engines, Unity and Unreal, with very little attention given to other options like Godot or proprietary engines, which also have unique strengths and use cases. Additionally, there is a lack of comprehensive comparisons regarding the choice of **GPU API**s (such as Vulkan, DirectX, or OpenGL) and how these impact engine performance and developer workflow. Only a few studies, like that by Szafran and Plechawska-Wójcik[10], briefly touch on the impact of graphics settings, but a detailed exploration of how different rendering pipelines and **GPU API**s influence game development remains largely unexamined.

These gaps highlight opportunities for future research to provide a more in-depth understanding of shader-specific development practices, the broader ecosystem of game engines, and the impact of various **GPU API**s and rendering pipelines on the overall development process.

Methodology and Data Preparation

To provide a comprehensive understanding of the current landscape of game engines and their role in modern game development, an extensive analysis was conducted using data from the **IGDB**. The **IGDB**, was chosen for its extensive API, which provides access to a large volume of information about released games. **IGDB** operates on a user-driven model, allowing users to contribute entries that are subsequently validated by either staff members or the wider user community. This ensures a high level of data accuracy and reliability while maintaining an expansive catalog of games across various platforms. This dataset encompasses all games registered as released on **IGDB** between October 18, 2019, and October 17, 2024. The chosen time frame captures recent trends while ensuring sufficient data volume for meaningful insights.

Data extraction was performed using a publicly available C# program (see Section Supplementary Material), ensuring transparency and reproducibility. The extracted dataset consists of the following fields:

- **first_release_date:** The game's initial release date in epoch

10. Szafran, Kamil, and Małgorzata Plechawska-Wójcik, Impact of Changes in Graphics Setting on Performance in Selected Video Games.

- **game_engines:** The game engine(s) used.
- **name:** The game's title.
- **platforms:** The platforms on which the game was released.

To illustrate the data structure, an example JSON object representing a game entry is provided in Listing 1. This structured approach allows for detailed exploration of trends across game engines and platforms. The full dataset is accessible at the provided supplementary material link.


```
{
    "id": 217028,
    "first_release_date": 1666569600,
    "game_engines": [
      {
        "id": 13,
        "name": "Unity"
      }
    ],
    "name": "Beat Blitz",
    "platforms": [
      {
        "id": 6,
        "name": "PC (Microsoft Windows)"
      },
      {
        "id": 14,
        "name": "Mac"
      },
      {
        "id": 34,
        "name": "Android"
      },
      {
        "id": 39,
        "name": "iOS"
      }
    ]
}
```


Listing 1. *example of JSON object pulled from IGDB*

Initially, discrepancies in engine naming conventions were observed, with entries including variations such as "Unity 2021" and "Unity3D." These inconsistencies complicated direct comparisons, necessitating a standardization process. Using a

Python script (See Section Supplementary Material), engine names were mapped to standardized labels, as shown in Table 1. This standardization enabled a unified analysis of engine popularity across platforms and provided a clearer picture of the distribution of game engines.

TABLE 1. *Standardized Game Engine Names*

Unstandardized Engine Labels	Standardized Game Engine Labels
unity, unity3d	Unity
unreal, ue	Unreal Engine
renpy, ren'py	Ren'Py
gamemaker, game maker	GameMaker Studio
godot	Godot
rpg maker	RPG Maker

Figures 1 and 2 illustrate the impact of this standardization. Figure 1 shows the fragmentation caused by unstandardized engine labels, with multiple variations of Unity and Unreal dominating the top ranks. This lack of consistency made it difficult to identify trends among less frequently used engines. After standardization (Figure 2), it becomes evident that while Unity and Unreal remain the most popular, many other engines also play significant roles in game development.

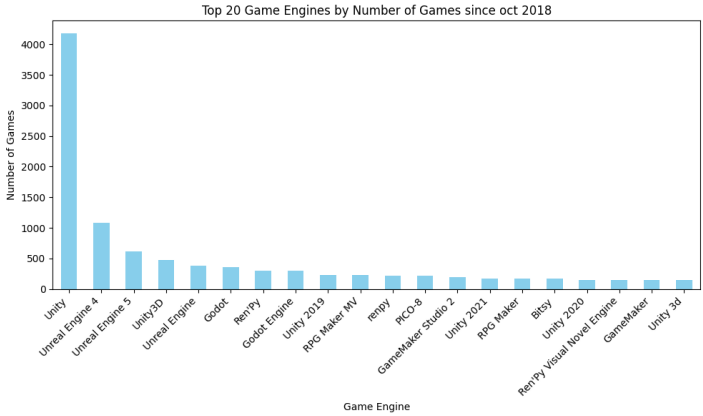
Engine Adoption and Graphics API Trends

The standardized dataset reveals six major engines as dominant: Unity, Unreal, RPG Maker, GameMaker, Ren'Py, and Godot. Together, these engines account for 10,551 of the 13,685 games in the dataset (77.1%), underscoring their central role in modern game development. This clarity, achieved through standardization, highlights the importance of properly structuring data for meaningful analysis.

Although GameMaker scores highly in terms of usage, it was excluded from further analysis due to significant differences between its versions, which complicate direct comparisons. Similarly, RPG Maker was excluded due to its reliance on HTML5 and OpenGL ES in newer versions, which differ substantially from the modern **API**s used by engines like Unity and Unreal.

Following this, an analysis of the graphics **API**s utilized by these engines was conducted, focusing on the platforms where the games were released and the corresponding graphics **API**s available. This analysis provides insight into how graphics **API**s, such as DirectX, Vulkan, Metal, and OpenGL, are leveraged across different environments.

The analysis excludes proprietary platforms such as PlayStation, Xbox, and Nintendo Switch in this specific section, as these use fixed **API**s—**Graphics Core Next Metal (GNM)** (PlayStation), DirectX (Xbox), and NVN (Nintendo Switch)—that



Notes:

FIGURE 1. *Unstandardized Engine Labels releases since Oct 2018*

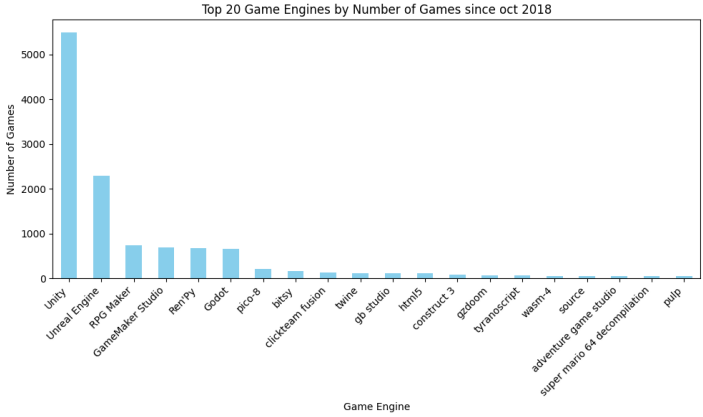


FIGURE 2. *Standardized Engine Labels releases since Oct 2018*

do not vary between engines. However, these platforms are revisited in later sections where the choice of shader languages (e.g., **HLSL** for DirectX or **PSSL** for **GNM**) becomes relevant to the discussion.

For clarity and to avoid unnecessary repetition, the full table of all platforms and **API**s is provided in Appendix A1. This ensures that readers interested in the broader context can reference the complete dataset.

Data collection was based on various sources, including official documentation, developer blogs, and third-party resources. For transparency, the specific sources for each platform and engine are cited directly below Table 2, ensuring that any assumptions or inferences can be verified through the original documentation.

TABLE 2. ***GPU API*** *used on different platforms*

Engine	Platform	**API**	Notes
Unity	PC (Microsoft Windows)	DirectX12, DirectX11, Vulkan, OpenGL, OpenGLES3	OpenGL is deprecated see[11]
Unity	Mac	Metal	
Unity	Linux	OpenGL, Vulkan[12]	
Unity	Android	Vulkan, OpenGLES[13]	
Unity	iOS	Metal[14]	
Unreal	PC (Microsoft Windows)	DirectX12, DirectX11, Vulkan, OpenGL	OpenGL is deprecated see[15]
Unreal	Mac	Metal	
Unreal	Linux	Vulkan, OpenGL	OpenGL is deprecated see[16]
Unreal	Android	Vulkan, OpenGLES	
Unreal	iOS	Metal	
Godot[17]	PC (Microsoft Windows)	DirectX12, Vulkan, OpenGLES	
Godot	Mac	Metal, OpenGLES	Uses Vulkan transpiled to Metal via MoltenVK.
Godot	Linux	Vulkan, OpenGLES	
Godot	Android	Vulkan, OpenGLES	

11. Unity Technologies, Unity OpenGL Deprecation and Removal for macOS and Windows.
12. Unity Technologies, Graphics APIs on Windows.
13. Unity Technologies, Graphics APIs on ios.
14. Unity Technologies, Graphics APIs on ios.
15. Epic Games, Warning: OpenGL is No Longer Supported for Desktop Platforms.
16. Epic Games, Warning: OpenGL is No Longer Supported for Desktop Platforms.
17. Godot Engine Developers, Rendering Drivers in Godot 4.

Engine	Platform	**API**	Notes
Godot	iOS	Metal, OpenGLES	Uses Vulkan transpiled to Metal via MoltenVK; OpenGL ES is available as a fallback for legacy devices.
GameMaker[18]	PC (Microsoft Windows)	DirectX11	
GameMaker	Mac	OpenGL	
GameMaker	Linux	OpenGL	
GameMaker	Android	OpenGLES	
GameMaker	iOS	OpenGLES	
RenPy[19]	PC (Microsoft Windows)	OpenGL	
RenPy	Mac	OpenGL	
RenPy	Linux	OpenGL	
RenPy	Android	OpenGLES	
RenPy	iOS	OpenGLES	

Graphics API Analysis and Observations

Unreal Engine employs a **Rendering Hardware Interface (RHI)**, which acts as an abstraction layer to facilitate support for multiple graphics **API**s, including DirectX, Vulkan, and Metal, across various platforms. This architecture provides developers with the flexibility to optimize rendering for specific hardware while maintaining a unified rendering pipeline.[20]

It is important to clarify that, Godot is described as using Metal on iOS and macOS, it actually relies on MoltenVK, a translation layer that converts Vulkan commands into Metal. This enables Godot to benefit from Vulkan's modern capabilities while maintaining compatibility with Apple's proprietary **API**.
The analysis of the graphics **API**s used across different platforms reveals several key trends:

The data for these observations are visualized in Figure 3. Each platform is represented with individual breakdowns, such as Figure 3a for Android, Figure 3b for iOS, and so on for each of the analyzed platforms. A legend is available in Figure 3f.

Android

As shown in Figure 3a, the usage of graphics **API**s is almost evenly split between Vulkan and OpenGLES, which makes sense given that Android platforms are generally Vulkan compatible, and Vulkan is the newer successor to OpenGL developed by the Khronos Group.

iOS

As depicted in Figure 3b, the graphics **API**usage is dominated by Metal, Apple's proprietary **GPU** API. This is expected, as both Unity and Unreal Engine, which are powerful and popular engines, leverage Metal to

18. Wikipedia Contributors, GameMaker Studio Graphics APIs.
19. The use of OpenGL and OpenGL ES in Ren'Py is not directly documented but is observable in the engine's open-source code on GitHub: Ren'Py Developers, n.d.
20. Epic Games, Render Hardware Interface (RHI) in Unreal Engine.

optimize performance on iOS devices. A small percentage is still openGLES, as this is techinically possible on some iOS devices though Godot, and the the fact that ren'py and GameMaker still uses OpenGLES.

Linux

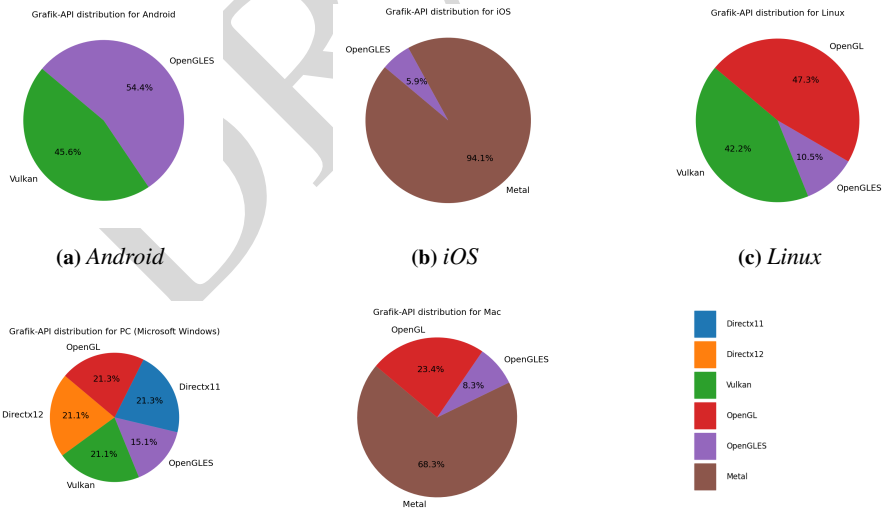
Referencing Figure 3c, the **API** usage on Linux is exclusively composed of Khronos Group technologies, specifically Vulkan and OpenGL. These **API**s are well known for their compatibility and native support on Linux platforms.

MacOS

As illustrated in Figure 3e, unlike iOS, macOS shows more diversity in **API** usage, with Vulkan and OpenGL being present alongside Metal. This could be attributed to the fact that more games are released on macOS through frameworks like Ren'Py and GameMaker, which traditionally rely on OpenGL.

Summary of API Observations

These observations provide valuable insight into the compatibility and adoption trends of graphics **API**s across different platforms. The distribution of **API**s reflects both the technical constraints of each platform and the strategic choices made by developers and game engines to ensure optimal performance and portability.



(a) *Android* **(b)** *iOS* **(c)** *Linux*

(d) *Windows* **(e)** *MacOS* **(f)** *Legend.*

FIGURE 3. *Distribution of Available **GPU APIs** depending on platform*

Impact of OpenGL Deprecation

To further understand the current trends regarding graphics **API** usage and the impact of deprecation decisions, the next step involved analyzing the effect of OpenGL's deprecation by both Unity and Unreal Engine on Windows and macOS platforms[21,22]. This analysis specifically focuses on these platforms, as OpenGL remains in use on other engines and platforms where newer **API**s such as Vulkan, Metal, or DirectX are unsupported or impractical.

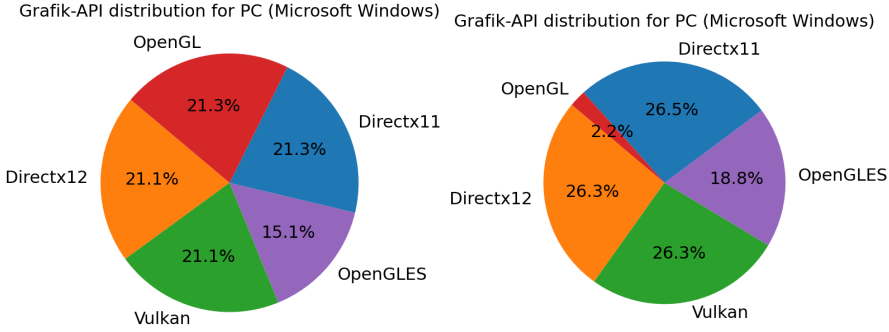
The exclusion of OpenGL aims to reveal how the distribution of graphics **API**s changes when this older **API** is no longer supported by these major engines. By comparing the original dataset, which included OpenGL as a supported API, with the revised data set that excludes OpenGL for Windows and macOS, this highlights notable changes in **API** preferences. This approach provides insights into the ongoing transition towards more modern graphics **API**s, such as Vulkan, Metal, and DirectX 12, and helps determine how these changes impact the distribution and adoption rates of different **API**s across platforms.

The revised results are illustrated in Figure 4 , which shows the adjusted distribution of graphics **API**s on the Windows platform without OpenGL for Unity and Unreal. Comparison of Figures 4a and 4b provides a clear visual representation of how OpenGL deprecation has changed the adoption of other **API**s, with a focus on understanding whether Vulkan or Metal has taken over the market share previously held by OpenGL. It also highlights the dominance of Unreal Engine and Unity, as this change heavily affected the distribution.

Focus on High-Fidelity Graphics Analysis

In this part of the analysis, the focus is specifically on engines that are capable of delivering high-fidelity, realistic 3D graphics. Consequently, engines such as Unreal, Unity, and Godot are considered for the subsequent detailed analysis. These engines are known for their robust capabilities in rendering complex 3D scenes and have widespread use in the game industry for producing visually striking content.

21. Unity Technologies, Unity OpenGL Deprecation and Removal for macOS and Windows.
22. Epic Games, Warning: OpenGL is No Longer Supported for Desktop Platforms.



(a) *Windows With OpenGL* **(b)** *Windows without OpenGL*

FIGURE 4. *Distribution **API** on Windows With And Without OpenGL*

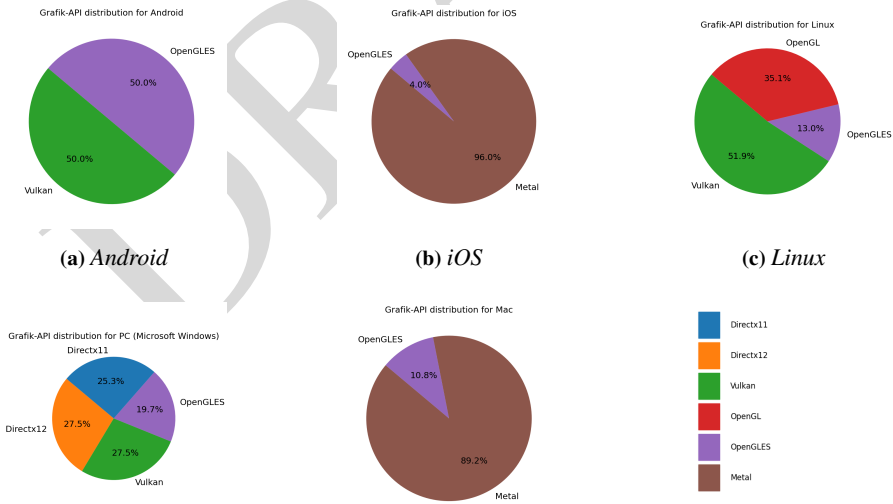
On the other hand, engines primarily used for 2D graphics or with limited support for realistic 3D rendering, such as RPG Maker, Ren'Py, and GameMaker Studio, are excluded from this analysis. These engines, while valuable for specific types of games, do not offer the same level of graphical fidelity needed for the kind of high-end visual experiences targeted in this study. The focus here is to assess the engines that are most relevant for developers aiming to produce immersive and state-of-the-art visual content in 3D.

To further understand the implications for graphics **API** usage, a similar comparative analysis was conducted, this time focusing solely on Unreal, Unity, and Godot. This revised analysis provides insights into the distribution of graphics **API**s across different platforms, specifically for engines that prioritize high-fidelity 3D graphics. The visualized data can be seen in Figure 5.

The revised analysis reveals some significant trends. On both Windows (Figur 5d) and macOS (Figure 5e) platforms, OpenGL has effectively been phased out and is no longer in use, indicating that OpenGL may not be suitable for high-fidelity graphics on these platforms. Instead, more specialized **API**s have taken its place—Metal is predominantly used for macOS, while DirectX versions and Vulkan dominate on Windows. This transition highlights the shift towards platform-optimized **API**s that provide better performance for complex 3D rendering tasks.

On Linux Figure (5c), however, OpenGL still plays a role, though Vulkan is increasingly being adopted, suggesting a gradual shift towards newer and more capable graphics **API**s. On Android it is a 50 percent split between opengles and vulkan indicating that even on this platform high fidelity graphics uses vulkan more.

The observations from this revised analysis underscore the ongoing evolution of graphics **API** adoption, driven by the need for higher performance and better platform optimization, particularly in engines used for high-end 3D graphics. This focus ensures that the findings are directly applicable to scenarios where achieving high visual fidelity is a core requirement.



(a) *Android* **(b)** *iOS* **(c)** *Linux*

(d) *Windows* **(e)** *MacOS* **(f)** *Legend.*

FIGURE 5. *Distribution of Available **GPU APIs** on high fidelity engines depending on platform*

Shader Language Analysis

The next part of the analysis focuses on the shader languages utilized by different **GPU APIs** across platforms. This analysis includes all the different **API**s even the propiertary ones, as most of these use existing and popular shading languages and not propiertary ones. To determine which shader languages are used for each **API**, a reference table 3 was created to map **GPU APIs** to their corresponding shader languages. It is worth noting that while Vulkan uses a binary format known as **Standard Portable Intermediate Representation for Vulkan (SPIR-V)**, shaders are typically authored in **GLSL** or **HLSL** and then compiled to **SPIR-V**. As such, Vulkan was counted for both **GLSL** and **HLSL** in this analysis.[23] This is true for NVM aswell.

TABLE 3. *Graphics **API**s and Corresponding Shader Languages*

API	Shader Language
OpenGL	**GLSL**
OpenGLES	**GLSL**
Metal	Metal Shading Language (**MSL**)
DirectX 11	**HLSL**
DirectX 12	**HLSL**
Vulkan	**HLSL /GLSL*** (**SPIR-V**)
GNM	**PSSL**
NVN	**HLSL /GLSL*** (**SPIR-V**)

Visualizations of the different distributions of shader languages, depending on the platform, provide further insights. The data, as illustrated in Figure 6, shows some interesting trends:

Windows and Xbox

as seen in figures 6j.6k and 6f **HLSL** clearly dominates on Microsoft-based platforms, which include Windows and Xbox. This is expected, given Microsoft's promotion of **HLSL** as the primary shading language for DirectX. The Windows platform still has a larger percentage of **GLSL**, this probably is due to the support of openGLES from Unity which uses **GLSL** and vulkans ability to use either **GLSL** or **HLSL** .

PlayStation Platforms

It is evident in figures 6g and 6h that PlayStation platforms primarily use PLSL, Sony's proprietary shader language, with a small portion marked as unknown. This is likely due to games released using the Godot engine, where the specific shader language can vary depending on how the game was ported[24]. It is still probable that PLSL is used in these cases.

Apple Platforms (macOS and iOS)

On Apple platforms (figures 6d and 6b, the Metal Shading Language (**MSL**) is dominant, with a small proportion of **GLSL** and **HLSL** , which can be attributed to Godot being able to use OpenGLES on both

23. Khronos Group, High-Level Shader Language Comparison.
24. Godot Engine Developers, Consoles — Godot Engine (stable) documentation in English.

platforms.

Nintendo Switch

The distribution is evenly split between **GLSL** and **HLSL** as seen in figure 6e. THis is due to fact that all the engines compiles to NVM which can use either **GLSL** or **HLSL**.

Linux and Android

Linux and Android show a similar distribution, as Vulkan or OpenGL can be used in all engines. This in seen in figures 6a and 6c.

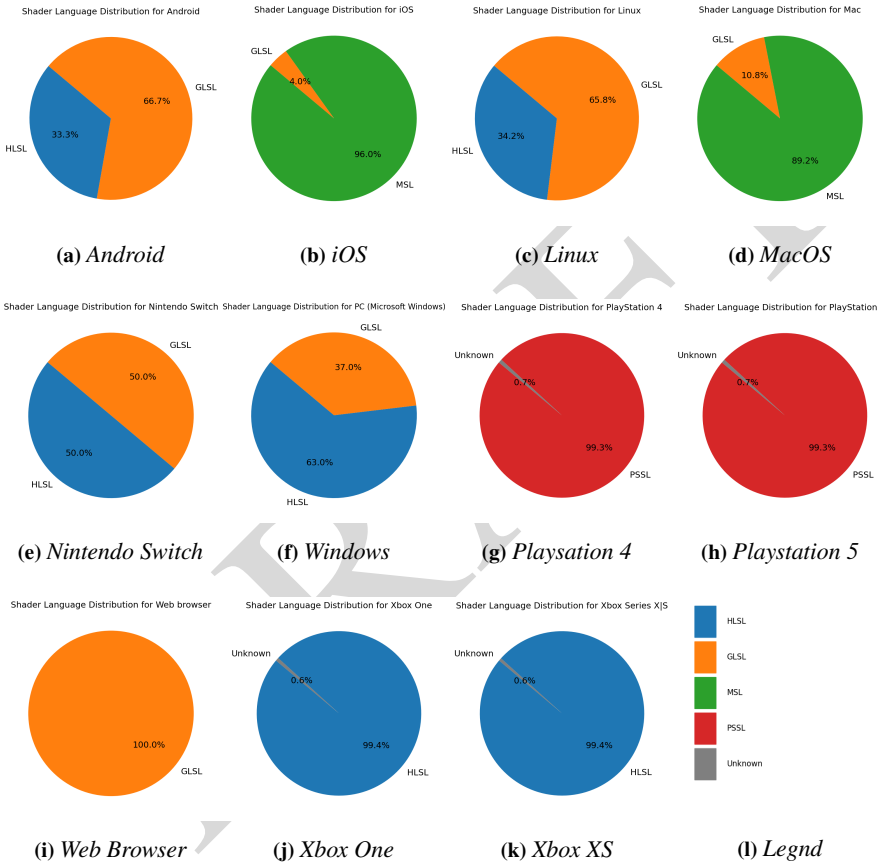
Web Browser

The web browser shading languages as seen in fingure 6i shows a 100% bias towards **GLSL**, as Webgl is the only supported web platform for the given engines at the time of writing.

Summary of Shading languages

The summary of these observations is that **HLSL** is clearly the most dominant shader language, this can be seen in 7. This is esspecially true factoring in that Vulkan and NVN can use both **GLSL** and **HLSL**. **PSSL** and **MSL** are also prevalent but are limited to their respective proprietary ecosystems (PlayStation and Apple). **GLSL** still remains relevant, and it is important to note that both **GLSL** and **HLSL** according to Anteru[25] are closely related C-style shader languages—knowledge of one can easily be transferred to the other. Similarly, Metal Shading Language (**MSL**), according to Galvan[26], is heavily influenced by other shader languages, such as **HLSL** and **GLSL**, and borrows concepts from both, making it accessible for developers familiar with these languages. PlayStation Shader Language (**PSSL**), on the other hand, while being similar to **HLSL**, has a closer alignment with PlayStation hardware features. According to Stenson[27], porting shaders from **HLSL** to **PSSL** is considered "fairly trivial," reflecting the syntactical and structural similarities between the two languages, albeit with extensions for PlayStation's unique capabilities.

25. Ludwig, Heiko, Mapping between HLSL and GLSL.
26. Galvan, Alain, A Review of Shader Languages.
27. Stenson, Richard, and Chris Ho, PlayStation Shader Language for PlayStation 4.



(a) *Android* **(b)** *iOS* **(c)** *Linux* **(d)** *MacOS*

(e) *Nintendo Switch* **(f)** *Windows* **(g)** *Playsation 4* **(h)** *Playstation 5*

(i) *Web Browser* **(j)** *Xbox One* **(k)** *Xbox XS* **(l)** *Legnd*

FIGURE 6. *Distribution of available shading languages based on platform built from high fidelity engines.*

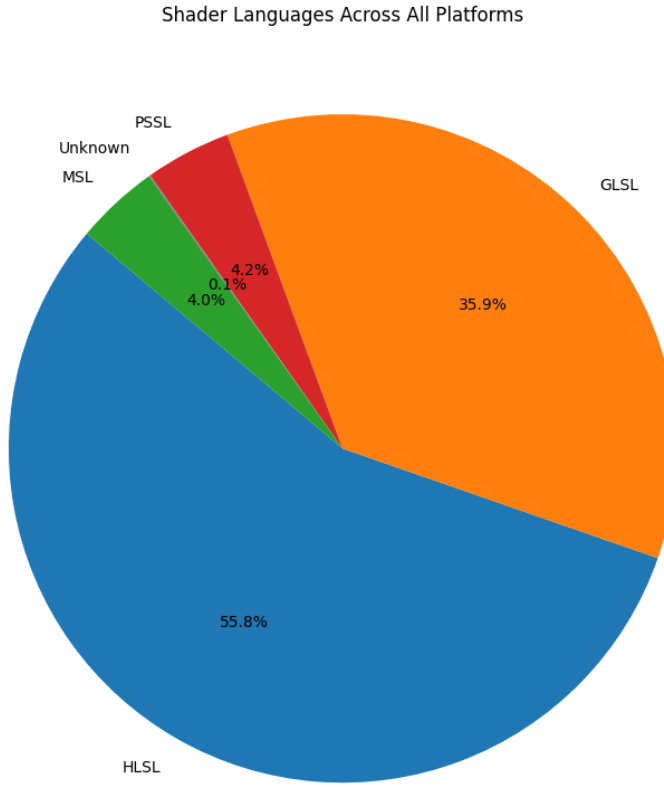


FIGURE 7. *Shader languages used on game releases since 18 Oct 2018*

Shader Development Tools and Workflows

The process of creating shaders differs significantly across various game engines, not only in terms of the language used but also in terms of the tools available for shader authoring. In Unity, developers can either use Shader Graph, a visual node-based editor, or write shaders in a variaty of languages directly. Shader Graph abstracts much of the complexity involved in shader authoring, making it accessible even to those with limited coding experience. In Unreal Engine, the Material Editor serves a similar purpose but with a different set of paradigms and optimizations for high-end visual fidelity. Godot offers a Shader Editor, which is somewhat of a hybrid between the two. In this section, each of these tools will be examined on how they influences the workflow, efficiency, and learning curve for developers.

Unity: Render Pipelines and Their Impact on Game Development

In the context of modern game development, Unity's render pipelines and the ability to customize shaders play a critical role in achieving the visual fidelity and flexibility required by high-end game engines. Unity provides several render pipelines—namely the Built-in Render Pipeline, **URP**, and **HDRP**—each tailored to specific graphical and hardware requirements. Additionally, Unity's **Scriptable Render Pipeline (SRP)** framework empowers developers to create and customize shaders, enabling further optimization of both graphics quality and performance.

Unity's implementation of render pipelines is designed to adapt rendering processes to various platforms and degrees of realism. The choice of render pipeline significantly impacts both the development process and the final visual output. This flexibility allows Unity to target complex 3D games as well as optimized projects for mobile and web platforms.

The following sections outline Unity's render pipelines, their influence on the development workflow, and the interaction between pipeline selection, graphics **API**s, and shader languages.

SRP

Unity introduced the **SRP** as a framework for developers to design customized rendering pipelines tailored to specific project needs. **SRP** provides developers with control over how graphical data is processed during the rendering stages, enabling bespoke solutions optimized for performance and visual fidelity. Both **URP** and **HDRP** are built on **SRP**, serving as standardized implementations that demonstrate the flexibility **SRP** provides.

Unity's Render Pipelines

Unity provides developers with three main render pipelines, each offering distinct advantages based on project requirements:

- **Built-in Render Pipeline:**
  Unity's legacy pipeline supports a broad range of platforms and offers flexibility for shader customization. While it lacks the modern features available in newer pipelines, it remains suitable for projects targeting PCs, consoles, mobile devices, and web platforms.
- **URP:**
  Designed to balance performance and visual fidelity, **URP** is optimized for scalability across mobile devices, PCs, and consoles. Although it supports advanced graphical features, it lacks some of the photorealistic capabilities found in **HDRP**.
- **HDRP:**
  HDRP caters to high-fidelity projects requiring advanced graphical features such as ray tracing and volumetric lighting. This pipeline is limited to powerful hardware platforms and does not support mobile devices or less capable consoles like the Nintendo Switch.

HDRP depends on modern **API**s to deliver its advanced rendering capabilities. For instance, it supports DirectX 12 on Windows, Vulkan on Linux, and Metal on macOS. While **HDRP** can run on DirectX 11, Unity strongly recommends DirectX 12 for optimal results, as **HDRP** is heavily optimized for the features offered by newer **API**s.

This pipeline setup allows developers to choose based on the desired balance between performance and visual fidelity, while the ability to customize shaders and materials remains consistent across Unity's render pipelines.

Graphics APIs and Render Pipelines

Unity's **SRP** and render pipelines are built with varying requirements for graphics **API**s, depending on platform capabilities. While OpenGL remains a supported **API**, primarily for older platforms and less demanding projects, Unity is shifting focus toward modern **API**s that enable advanced graphical effects and optimized performance.

- **URP** and the Built-in Pipeline continue to support OpenGL for mobile devices and web platforms. However, high-fidelity effects in advanced projects benefit from modern **API**s such as DirectX 12 (Windows), Vulkan (Linux), and Metal (macOS).
- **HDRP** requires modern **API**s capable of supporting advanced features like ray tracing and volumetric lighting. As a result, OpenGL is not supported for **HDRP** projects.

Unity's gradual move away from OpenGL reflects the industry's broader focus on photorealistic graphics and performance optimization for modern hardware[28].

Rendering Modes in Unity's Render Pipelines

Unity's render pipelines support various rendering modes, each tailored to specific requirements for lighting, shadowing, and performance:

- **Forward Rendering:**
  Efficient for scenes with a small number of light sources, Forward Rendering calculates lighting on a per-object basis, making it suitable for less complex lighting setups.
- **Deffered Rendering**
  Designed for scenes with multiple light sources, Deferred Rendering processes lighting in a separate pass, allowing efficient handling of complex lighting setups. This mode is particularly relevant for **HDRP** projects.
- **Forward+ Rendering**
  Primarily available in **HDRP**, Forward+ Rendering extends the capabilities of Forward Rendering, supporting a larger number of dynamic light sources with minimal performance impact.

These rendering modes influence how light and shadows are calculated and play a key role in creating custom shaders.

Future Developments in Unity's Render Pipelines

Unity has indicated that future development efforts will prioritize **URP** and **HDRP**. In Unity 6, approximately 90% of new PC and console projects are expected to utilize either **URP** or **HDRP**, with the Built-in Pipeline receiving support for at least two years post-launch. This shift highlights Unity's focus on optimizing **URP** and **HDRP** for modern platforms and phasing out legacy systems.[29]

Custom shaders in Unity

In Unity, materials act as a bridge between shaders and objects in a scene. A material defines the surface appearance by linking a shader and adjusting its properties. Developers can customize materials with specific parameters, such as colors, textures, and reflection properties, providing flexibility to create diverse visual effects without modifying the shader code itself. Materials allow the reuse of a single shader across

28. Technologies, Unity, Unity's Future with Render Pipelines.
29. Technologies, Unity, Unity's Future with Render Pipelines.

multiple objects with different appearances, simplifying the process of achieving visual variations in a project.

Unity offers two primary approaches for working with shaders: Shader Graph and code-based shaders. Shader Graph is a visual, node-based tool that simplifies the creation and editing of shaders, particularly for the **URP** and the **HDRP**. Although originally designed for **URP** and **HDRP**, Shader Graph is also compatible with the Built-in Render Pipeline. However, Shader Graph for the Built-in Pipeline only receives bug fixes and no new features, as Unity's focus has shifted to **URP** and **HDRP**[30].

Code-Based Shaders

For code-based shaders, Unity supports several languages, including **HLSL**, **C for Graphics (CG)**, and **GLSL**, depending on the target platform and chosen graphics API. Unity recommends using **HLSL** for **URP** and **HDRP**, as these pipelines are optimized for it[31]. **CG**, while supported, is mainly used for legacy projects in the Built-in Pipeline and does not benefit from the advanced optimizations available in **SRP** pipelines[32]. Additionally, Unity advises against using **GLSL** except for testing purposes due to its limited integration with the engine's rendering systems[33].

Unity's shader pipeline is orchestrated through ShaderLab, a domain-specific language unique to Unity. ShaderLab acts as a wrapper and configuration system, providing metadata and directives that connect Unity's rendering engine to the underlying shader code. It is within ShaderLab files that developers define SubShaders, which specify multiple fallback shaders, as well as Passes, which dictate the rendering pipeline stages the shader should engage with. For example, a SubShader in ShaderLab might include a high-fidelity pass for modern platforms and a simplified pass for legacy devices.

The key role of ShaderLab includes:

1. **Render Queue Control:** Developers can specify when objects are rendered relative to others, such as opaque or transparent objects.
2. **State Management:** ShaderLab configures blending modes, depth testing, and stencil operations, enabling precise control over the rendering pipeline without modifying shader code directly.
3. **Multiple Passes:** Developers can define multiple rendering passes within a single shader, allowing for advanced effects like multi-layered rendering or shadow mapping.

An example ShaderLab can be seen in listing 2
Here, ShaderLab provides essential configuration (e.g., RenderType and Fallback) while the actual shader code is implemented in **HLSL**, **CG**, or **GLSL**. When creating shaders in Unity, the build system automatically compiles the shader code into the optimal format for the target **API** and platform. For example, **HLSL** is compiled for DirectX 12 on Xbox and Windows, while **GNM** is used for PlayStation 5. This ensures that shaders function seamlessly across platforms without requiring developers to manually adapt them for each API[34]. Typically, shaders are compiled into machine code only when executed on the target platform, providing flexibility since the **GPU** specifics are only identified during runtime[35].

ShaderLab also manages these cross-platform complexities. By organizing SubShaders and Passes, it allows Unity to automatically select the most appropriate rendering path for the current platform, ensuring consistent performance and visuals.

Despite Unity's flexibility in supporting multiple shader languages, **HLSL** has emerged as the preferred choice for **URP** and **HDRP** due to its robust optimization capabilities and compatibility with modern hardware[36]. **GLSL**, as outlined in Unity's documentation, is generally not recommended for production

30. Unity Technologies, Choosing a Render Pipeline.
31. Unity Technologies, Choosing a Render Pipeline.
32. Unity Technologies, Writing Shaders for the Universal Render Pipeline.
33. Unity Technologies, GLSL Shader Programs.
34. Unity Technologies, Render Pipelines Feature Comparison.
35. Unity Technologies, Render Pipelines Feature Comparison.
36. Unity Technologies, Choosing a Render Pipeline.

```
Shader "Example/BasicShader"
{
    SubShader
    {
        Tags { "RenderType" = "Opaque" }
        Pass
        {
            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            void vert() { /* Vertex Shader Code */ }
            void frag() { /* Fragment Shader Code */ }
            ENDHLSL
        }
    }
    Fallback "Diffuse"
}
```

Listing 2. *Example of a basic ShaderLab/HLSL shader in Unity.*

use[37]. CG is retained for backward compatibility but lacks compatibility with newer pipelines and their advanced features[38].

Shader Graph

Shader Graph is Unity's node-based tool for creating shaders visually without writing code. This tool simplifies working with advanced effects and material properties, making it accessible even for developers with no experience in **HLSL** or ShaderLab. Shader Graph integrates tightly with **URP** and **HDRP**, but for the Built-in Pipeline, it only receives bug fixes and no new features[39].

Shaders in Shader Graph are constructed using nodes, which can be combined to create complex effects. For larger projects, developers can create Subgraphs, which can be saved and reused in other Shader Graphs, improving organization and maintainability. Shader Graph also supports Custom Function Nodes, enabling the inclusion of **HLSL** code to extend functionality. This is particularly useful for special customizations or approximations that cannot be achieved or optimized using standard nodes.

Shader Graph manages its integration with Unity's rendering systems and lighting models automatically, ensuring that shaders interact seamlessly with scenes and light sources. It supports the rendering modes of **URP** and **HDRP** (e.g., forward and deferred rendering), which means that lighting and shadow calculations

37. Unity Technologies, GLSL Shader Programs.
38. Unity Technologies, Writing Shaders for the Universal Render Pipeline.
39. Unity Technologies, Choosing a Render Pipeline.

are handled without requiring additional configuration. This feature is especially advantageous in **HDRP**, where Shader Graph leverages advanced lighting models that can be challenging to implement manually.

Balacing Visual workflow and Code

Shader Graph offers an intuitive visual workflow but also includes options for extending functionality through Custom Function Nodes, which enable the addition of **HLSL** code. By referencing external **HLSL** files, developers can modify parameters such as brightness, color, or lighting effects while still leveraging Unity's existing rendering pipeline and lighting systems. This hybrid approach combines the ease of visual construction with the precision of coding.

However, Custom Function Nodes are subject to certain limitations. They rely on Unity-specific syntax, and parameters must be defined and referenced with precise string matches in Shader Graph's interface. This dependency on Unity's conventions can make code less flexible and more prone to errors if parameters are mismatched. While Custom Function Nodes do not offer the same level of freedom as pure **HLSL**, they provide a balanced approach, enabling visual workflows to coexist with Unity-specific coding conventions.

Although Shader Graph provides ease of use and visual construction, it has limitations in terms of low-level optimization and hardware-specific tailoring compared to **HLSL**-based shaders. Shader Graph depends on Unity's internal rendering pipeline, making it less flexible than **HLSL** for targeting specific hardware features or graphics **API**s.

Unreal: Rendering And Presets

Unreal Engine uses a unified rendering pipeline, augmented by rendering presets that adapt the engine's behavior to different platforms and hardware capabilities. These presets configure the rendering features by toggling options and scaling their complexity to balance performance and visual fidelity. Unlike Unity's approach with distinct render pipelines, Unreal Engine uses presets to optimize rendering settings while maintaining a single, flexible pipeline.[40]

Preset Types

Unreal Engine organizes rendering presets along two main dimensions:

- **Target Platform:**
  - **Desktop and Console:** Designed for high-performance systems, enabling advanced features such as Lumen (dynamic global illumination) and Nanite (virtualized geometry). These platforms prioritize photorealism and high graphical fidelity.[41]
  - **Mobile:** Scaled for smartphones and tablets, where performance takes precedence. Many high-end features are reduced in complexity or replaced with more efficient alternatives to accommodate hardware limitations.
- **Quality Preset:**
  - **Maximum Quality:** Activates high-end graphical features, such as real-time lighting and high-resolution textures, making it ideal for premium hardware. For example, platforms supporting **Shader Model (SM)** 6+ enable features like Nanite and advanced Lumen settings.
  - **Scalable Quality:** Disables or simplifies features to ensure compatibility with less powerful hardware. This includes relying on baked lighting instead of real-time lighting or reducing shadow resolution.

40. Epic Games, Supported Features by Rendering Path for Desktop with Unreal Engine.
41. Epic Games, Nanite Virtualized Geometry in Unreal Engine.

These presets provide a flexible starting point for balancing performance and quality, particularly for multi-platform projects.

How Rendering Presets Work in Unreal Engine

Rendering presets are implemented using Device Profiles, which determine the default settings for specific platforms. These profiles dynamically adjust engine features by toggling console variables that define Unreal's rendering capabilities.[42] For example:

- On high-end platforms, Lumen may run at full resolution with real-time dynamic lighting, while on mobile it may revert to baked lighting or lower-quality approximations.
- Nanite, reliant on **SM** 6+, is automatically disabled on platforms that do not support this **SM**, such as older Vulkan configurations.[43]

These profiles ensure that projects can scale automatically across devices, although developers can manually override these settings for more granular control.

Rendering Modes in Unreal Engine

Unreal Engine primarily uses **Deferred Rendering** as its default mode, optimized for high-fidelity projects with complex lighting setups. Unlike Unity, where different pipelines (e.g., URP for forward rendering and HDRP for deferred) dictate the rendering mode, Unreal offers a unified pipeline that dynamically adjusts based on the project's requirements.

Deferred rendering in Unreal integrates seamlessly with advanced features like **Lumen** and **Nanite**, making it suitable for PC and console platforms where performance and visual fidelity are priorities. For scenarios requiring lower overhead, such as VR or mobile platforms, Unreal supports **Forward Rendering** and **Forward+**, enabling efficient management of dynamic lights through clustered lighting techniques.

While developers have the flexibility to choose a rendering mode, the engine's presets and device profiles often determine the optimal mode automatically, streamlining the workflow. This ensures that Unreal's rendering modes are both adaptable and deeply integrated with its core systems.

Customization and Overrides

While presets are an efficient way to manage rendering, developers retain the ability to customize them. For instance:

- Device Profiles can be modified directly to adjust settings for a specific device or platform. This includes enabling or disabling Lumen Reflections or setting fallback options for unsupported hardware features.
- Developers can override presets at runtime to provide end-users with options, such as toggling between "Performance Mode" and "Cinematic Mode" on consoles.

These overrides highlight the flexibility of Unreal's rendering architecture while allowing developers to fine-tune settings for specific scenarios.

Challenges and Documentation Gaps

While Unreal Engine's rendering presets are robust, understanding the interaction between presets, device profiles, and platform-specific **GPU APIs** can be complex. For example:

42. Epic Games, Customizing Device Profiles and Scalability in Unreal Engine Projects for Android.
43. Epic Games, Nanite Virtualized Geometry in Unreal Engine.

- Nanite and Lumen are heavily dependent on **GPU** capabilities, such as support for **SM** 6+. If fallback solutions are not implemented, developers risk performance bottlenecks or visual artifacts on unsupported hardware.[44]
- Documentation gaps: While Unreal's documentation provides overviews of supported features across rendering paths, details about specific interactions and configurations across all platforms remain sparse. Developers often rely on community resources or trial and error for deeper insights.

Despite these challenges, Unreal's rendering presets provide a powerful mechanism for scaling visual quality and performance across diverse platforms.

RHI: Translating Rendering Code to GPU APIs

Unreal Engine's **RHI** is a critical abstraction layer that ensures compatibility between Unreal's rendering systems and a wide range of **GPU APIs**. By acting as a translator, **RHI** simplifies cross-platform development and ensures that rendering features function correctly, regardless of the underlying hardware and **API**.

The Role of RHI in Shader Compilation and APITranslation

The **RHI** automates the process of adapting Unreal Engine's rendering features to the target platform's **GPU API**. This involves several steps:

1. **Shader compilation:** Unreal shaders, written in **HLSL** or created visually in the Material Editor, are first processed by the **RHI**. The **RHI** ensures that shaders are compiled into the appropriate format for the target API:[45]
   - DirectX shaders are compiled into **HLSL** bytecode.
   - Vulkan shaders are converted into **SPIR-V**, Vulkan's native shader language.
   - Metal shaders are translated into Metal Shading Language (**MSL**).
   - OpenGL/OpenGL ES shaders use **GLSL**.

   This process ensures that shaders written once in Unreal can run seamlessly on multiple platforms without requiring manual adaptation by developers.

2. **Runtime Adaptation:**
   During runtime, **RHI** dynamically selects the appropriate **API** based on the platform and hardware capabilities.
   For example:
   - On a high-end Windows PC, **RHI** may default to DirectX 12 to leverage ray tracing and other advanced features.
   - On mobile, **RHI** might fall back to OpenGL ES if Vulkan is unavailable or unsupported by the hardware.

   This automatic adaptation allows Unreal to scale its rendering features while maintaining compatibility with older or less capable hardware.
3. **Resource Management:**
   - On Vulkan, **RHI** manages descriptor sets and ensures efficient use of **GPU** memory.
   - On Metal, **RHI** optimizes the rendering pipeline to align with Apple's architecture.
4. **Feature Fallbacks:**
   - A project using Lumen's dynamic lighting may fall back to baked lighting on hardware that lacks real-time global illumination support.
   - Nanite's virtualized geometry automatically switches to traditional polygon-based rendering on unsupported platforms.

44. Epic Games, Supported Features by Rendering Path for Desktop with Unreal Engine.
45. Epic Games, Rendering Hardware Interface (RHI) Overview.

Advantages of RHI in Cross-Platform Development

RHI provides several key benefits for developers:

- **Platform-Agnostic Development:** Developers can focus on creating high-quality visuals without worrying about the intricacies of each **GPU** API. **RHI** ensures that the same rendering code works across all supported platforms.
- **Unified Shader Workflow:** Whether using Material Editor or HLSL, developers write shaders once, and **RHI** ensures they are compiled correctly for each target platform.
- **Scalability: RHI**'s dynamic adaptation allows Unreal to deliver consistent performance and visuals across a wide range of devices, from mobile phones to high-end gaming PCs.

Limitations and Developer Considerations

While **RHI** simplifies many aspects of rendering, it also introduces certain challenges:

- **Performance Overheads:** Abstracting API-specific details can introduce minimal overhead. Advanced developers may need to fine-tune settings for optimal performance on specific platforms.
- **Fallback Complexity:** Developers targeting a wide range of hardware must implement fallback solutions to handle features like Lumen and Nanite on unsupported platforms.

Practical Example: Shader Compilation Pipeline

To illustrate **RHI**'s functionality, consider a shader written in **HLSL** for a project targeting multiple platforms:

1. The shader is authored using Unreal's Material Editor or written in **HLSL** for specific effects.
2. During the packaging process, **RHI** compiles the shader:
   - Converts **HLSL** to DirectX bytecode for Windows platforms.
   - Translates **HLSL** to **SPIR-V** for Vulkan (e.g., Android or Linux).
   - Converts the shader into **MSL** for Apple's Metal API.
3. At runtime, **RHI** ensures the shader is executed correctly by the target **GPU API**, adapting to platform-specific constraints like texture formats or memory layouts.

Shader Workflow in Unreal Engine

In Unreal Engine, materials serve as the fundamental bridge between the visual appearance of an object and the underlying shader code. Materials dictate how a surface interacts with light, including properties such as color, reflectivity, bumpiness, and transparency. These surface properties are determined by combining textures, Material Expressions, and configurable settings within the Material Editor. In essence, materials define every visual aspect of a surface, guiding the render engine on how to process lighting and shading for objects in the scene.

Creating Materials with the Material Editor

The Material Editor is Unreal's primary tool for creating shaders visually through node-based Material Expressions. These expressions are translated into **HLSL** shaders during compilation, which are subsequently optimized for specific **GPU** architectures. Developers work with the main material node, defining parameters such as blend modes and shading models. For instance, changing the blend mode from Opaque to Translucent unlocks additional parameters, like opacity control, that are not available in the opaque configuration[46].

46. Epic Games, Essential Unreal Engine Material Concepts.

One of the most powerful features of the Material Editor is its ability to preview the generated shader code. While this code is read-only, it provides insights into the complexity of Unreal's integrated shading systems. For example, a default-lit opaque material can generate around 5000 lines of **HLSL** code due to the inclusion of Unreal's advanced lighting and reflection models. This tightly integrated architecture ensures that developers focus primarily on material inputs, while Unreal handles the complexities of rendering, including multi-platform quality settings and **SM** optimizations.

Platform-Specific Optimization and Scalability

The Material Editor includes tools like Platform Stats, allowing developers to analyze shader instruction counts for different configurations, such as Vulkan Mobile **SM**5 or DirectX PC **SM**6. This provides a direct way to assess the resource usage of materials across various platforms. Unreal's scalability system further supports performance optimization by automatically toggling features or adjusting quality levels based on the target platform. For instance, a material designed for high-end PC rendering might disable certain real-time lighting features when running on mobile hardware. This flexibility ensures that shaders scale appropriately without manual intervention[47].

Reusability with Material Functions

To simplify shader workflows, Unreal Engine supports Material Functions, which encapsulate reusable combinations of Material Expressions. These functions streamline shader development, enabling developers to apply shared logic across multiple materials. Unreal provides a library of standard Material Functions that can be explored and extended for specific use cases, enhancing both productivity and consistency across projects.

Extending the Material Editor with Custom HLSL

For scenarios where advanced functionality is required, Unreal provides a Custom Expression node within the Material Editor. This node allows developers to write **HLSL** code directly, defining custom inputs and outputs. However, the Custom Expression node comes with certain limitations:

- It does not support constant folding, an optimization Unreal uses to reduce shader instruction calls automatically.
- Variables cannot be mutated directly; instead, the node requires explicit return values for all calculations.
- Newly created structs cannot include parameters unless defined at the outermost scope.

While the Custom Expression node offers flexibility, it is recommended for use only when necessary functionality cannot be achieved with existing Material Expressions. Developers can leverage Platform Stats to ensure that the custom logic does not introduce significant performance overhead[48].

Integration with Unreal's Rendering Systems

Unreal's Material Editor seamlessly integrates with its rendering architecture, enabling developers to focus on inputs while Unreal manages the underlying complexities. This includes dynamic adaptation to platform-specific constraints and multi-quality settings, such as low, medium, high, and epic, which allow features to be toggled based on **SM**s or device profiles. For example, a material using Lumen for dynamic global illumination might automatically fallback to baked lighting on unsupported hardware, demonstrating Unreal's robust scalability systems.

47. Epic Games, Unreal Engine Material Editor UI.
48. Epic Games, Custom Material Expressions in Unreal Engine.

Summary of Unreal Shader Workflow

Unreal Engine's Material Editor provides an efficient, scalable framework for creating and managing shaders. By combining visual workflows with the ability to extend functionality through HLSL, developers can achieve a balance between customization and ease of use, ensuring optimized performance and visual fidelity across diverse platforms.

Rendering in Godot Engine

Godot Engine employs a forward and forward+ rendering architecture, tailored to deliver optimal performance and visual quality across a diverse range of platforms. Unlike engines like Unreal and Unity, which include deferred rendering pipelines for high-end scenarios, Godot focuses on forward and forward+ to maintain simplicity, memory efficiency, and compatibility. This decision aligns with Godot's commitment to scalability and accessibility for developers.[49]

Renderer Options and Compatibility

When starting a new project, Godot provides three renderer options:

- **Forward+ (Standard):** Optimized for high-performance platforms, using the RenderingDevice abstraction to support modern **API**s like Vulkan, DirectX 12, and Metal (via MoltenVK). This is the default renderer for most projects, offering clustered lighting and advanced effects.
- **Mobile Renderer:** A lightweight variant designed for low-power and mobile devices, also using the RenderingDevice abstraction. It prioritizes performance over visual fidelity, scaling down effects where needed.
- **Compatibility Renderer:** Uses OpenGL exclusively (OpenGL 3.3 on desktop, OpenGL ES 3.0 on mobile). It is intended for legacy hardware, lacking clustered lighting and advanced effects available in other renderers.

The Compatibility renderer bypasses the RenderingDevice abstraction, relying solely on OpenGL for rendering commands. In contrast, the Forward+ and Mobile renderers utilize the RenderingDevice to ensure cross-platform compatibility and leverage modern **GPU** features.

Forward and Forward+ Rendering

Godot utilizes forward rendering for simpler lighting setups, particularly on mobile and low-end hardware, where performance and compatibility are paramount. Forward rendering calculates lighting directly in the fragment shader, limiting the number of dynamic lights but keeping computational costs low.

Forward+ builds upon this approach with clustered lighting. By dividing the screen into clusters and associating lights with each cluster, forward+ supports numerous dynamic light sources more efficiently. This method is particularly advantageous on platforms capable of handling **SM** 4.5 or higher, enabling Godot to scale its lighting complexity based on hardware capabilities.

Unlike Unity's forward+ implementation, which integrates deferred-like features, Godot's forward+ is streamlined for performance. This ensures compatibility across Vulkan, DirectX 12, OpenGL, and other supported **API**s.

Why Godot Avoids Deferred Rendering

Godot explicitly avoids deferred rendering due to its higher memory requirements and the complexity of implementing G-buffer storage. Deferred rendering is advantageous for scenes with numerous dynamic

49. Godot Engine Documentation Team, Internal Rendering Architecture.

light sources, but it comes at the cost of increased bandwidth and memory usage. These trade-offs conflict with Godot's focus on being lightweight and accessible across a wide range of devices.

By relying on forward and forward+, Godot avoids the overhead of deferred rendering while still supporting multiple light sources effectively. This decision simplifies the rendering pipeline and reduces the burden on lower-end and mobile devices.

GPU API Support and RenderingDevice Abstraction

Starting with Godot 4.0, Vulkan became the default rendering backend, marking a significant shift from OpenGL ES 3.0 and OpenGL ES 2.0. Vulkan enables advanced features such as clustered forward+ lighting and real-time global illumination while providing superior **GPU** utilization.

To ensure compatibility with non-Vulkan platforms, Godot leverages MoltenVK for Metal **API** support on macOS and iOS. Additionally, Godot now includes support for DirectX 12, broadening its reach to high-performance Windows systems. OpenGL remains available for legacy devices, particularly with OpenGL ES 3.0 and ES 2.0 for mobile and older hardware.

Central to this multi-**API** support is Godot's **RenderingDevice** abstraction, which decouples rendering code from specific **GPU API**s. This abstraction layer provides a unified interface for developers, while the engine handles the translation of rendering commands to the target API. For example:

- Vulkan commands are mapped directly for platforms supporting Vulkan natively.
- Metal compatibility is achieved via MoltenVK, translating Vulkan commands to Metal.
- DirectX 12 commands are generated for Windows platforms requiring high-end graphical features.
- OpenGL commands are used for devices with legacy hardware or limited Vulkan support.

This approach ensures that Godot's rendering pipeline can adapt seamlessly across platforms while preserving a consistent development experience.

Lighting and APIConstraints

Godot's **GPU API**support influences the lighting capabilities available on different platforms:

- **Vulkan:** Fully supports clustered forward+ lighting, enabling advanced effects like real-time global illumination and high-quality shadows.
- **DirectX 12:** Provides parity with Vulkan for advanced features on Windows platforms, including support for clustered lighting and modern **GPU** optimizations.
- **Metal (via MoltenVK):** Enables Vulkan-based features on macOS and iOS, albeit with slight performance trade-offs.
- **OpenGL ES 3.0:** Supports basic forward rendering with dynamic lighting but lacks the clustered lighting capabilities of Vulkan and DirectX 12.
- **OpenGL ES 2.0:** Limited to minimal lighting features, suitable for legacy devices requiring basic rendering pipelines.

The RenderingDevice abstraction ensures these features are dynamically adjusted based on platform constraints, with Godot developers focusing on high-level material and shader definitions rather than low-level **API** management.

Shader Workflow in Godot Engine

Godot Engine provides two primary methods for creating shaders: code-based shaders written in Godot's custom shader language and node-based shaders created in the Visual Shader Editor. Each approach offers flexibility and scalability, allowing developers to optimize workflows based on project requirements.

Custom Shaders in Godot

Godot's custom shader language is designed to be accessible yet powerful, closely resembling **GLSL** in syntax and functionality. This language simplifies shader development by automating many tasks, including:

- Declaring vertex attributes such as position, normal, and UV coordinates.

- Calculating the final vertex position in clip space.
- Passing interpolated attributes like UV coordinates and normals from the vertex shader to the fragment shader.

These default operations mean that leaving the vertex shader empty results in standard transformations being applied automatically. However, developers can override or modify these defaults to implement custom transformations or behaviors.

Godot's shaders follow a three-stage pipeline: **vertex**, **fragment**, and **light**. Data can be passed between stages using `varying` variables, which allow developers to send interpolated values from the vertex to the fragment shader, and from there to the light shader. For example, normals can be transformed into world space in the vertex shader and accessed in the light shader for custom lighting calculations.
Shader types in Godot include:

- **Spatial Shaders:** For 3D rendering, supporting advanced effects like custom lighting models.
- **CanvasItem Shaders:** For 2D rendering, targeting UI elements or sprites.
- **Particle Shaders:** For controlling particle effects.
- **Sky and Fog Shaders:** Specialized shaders for atmospheric effects.

Visual Shaders in Godot

The Visual Shader Editor in Godot offers a node-based interface for creating shaders, making it easier for developers without extensive coding experience. It consists of three main output nodes: **Vertex**, **Fragment**, and **Light**, which correspond to the stages in Godot's rendering pipeline.

Developers can use the **VaryingSetter** node to define and transfer custom interpolated values between stages. For instance, a varying variable defined in the vertex shader can be interpolated and used in both the fragment and light stages.

For advanced customization, the **Expression** node allows developers to write Godot shader language code within the visual workflow. This node supports custom inputs and outputs, enabling developers to combine the flexibility of code with the simplicity of visual workflows.

Balancing Code-Based and Visual Shaders

Godot's shader workflows provide developers with a choice between code-based and visual approaches, each suited to different use cases:

- **Code-Based Shaders:** Ideal for projects requiring precise control over rendering or advanced effects. The streamlined syntax and automatic operations reduce boilerplate code, allowing developers to focus on customization.
- **Visual Shaders:** Useful for rapid prototyping or when collaborating with non-programmers, such as artists. The Visual Shader Editor's intuitive interface makes it easy to iterate on designs while still supporting custom logic through the Expression node.

Godot's flexibility ensures that both methods integrate seamlessly with the engine's rendering systems, allowing developers to balance ease of use with the power of custom shader development.

Comparison of Shader Workflows Across Engines

Shader workflows vary significantly between Unreal Engine, Unity, and Godot, especially when comparing how developers interact with code-based shaders, node-based visual shaders, and engine-specific rendering systems.

Code-Based Shaders comparison

Unreal Engine

Writing custom **HLSL** shaders in Unreal Engine is often considered complex and impractical for most workflows. While the engine provides significant power and flexibility, developers must understand Unreal's

intricate rendering pipeline, including the **RHI** and shading models. Hooking into existing lighting systems or extending default behaviors generally requires an in-depth knowledge of the engine's internals.

According to a discussion by user Shadowriver on the Unreal Forums, the process of writing custom **HLSL** shaders in Unreal Engine 4 is particularly challenging, given that the engine lacks built-in support for custom **HLSL** shaders as a core feature. Instead, developers must rely on unconventional workarounds, such as modifying engine source code or integrating custom rendering passes.[50] While it is possible that some of these limitations have been addressed in Unreal Engine 5, no official documentation confirms significant improvements in this regard.

For these reasons, most developers rely on the Material Editor, which abstracts shader complexity and integrates seamlessly with Unreal's rendering systems. Writing custom **HLSL** shaders is therefore typically reserved for highly specific or experimental use cases due to the steep learning curve and technical challenges involved.

Unity

Unity offers a more accessible approach to code-based shaders through its ShaderLab abstraction. ShaderLab provides a unified interface for defining shader properties, passes, and interactions with Unity's rendering systems. Developers write shaders in **HLSL**, **CG**, or **GLSL**, with ShaderLab handling the platform-specific implementation details.

One key advantage of Unity's approach is the ability to leverage low-level features for shader optimization. Developers can set custom compiler flags, target specific **SM**s, and fine-tune behavior for different hardware configurations. For example, ShaderLab allows advanced users to optimize shaders by specifying separate passes for forward or deferred rendering, which makes it a powerful tool for developers familiar with traditional rendering pipelines. **HLSL** is recommended for SRP variant pipelines, ensuring compatibility and optimal performance.

While ShaderLab simplifies shader creation, integrating with existing systems like Unity's built-in deferred lighting model requires a deep understanding of Unity's rendering architecture. Developers must manually adapt their shaders to interact with the G-buffer and other engine-specific components. This complexity can make high-level integrations challenging compared to Godot, but it provides significant flexibility for optimizing low-level performance by using compiler tags directly in the code.

However, Unity's Shader Graph offers a streamlined alternative for creating and integrating shaders. Shader Graph enables developers to visually design materials while seamlessly integrating with Unity's existing lighting models and rendering features. Through the use of a Custom Node, developers can also write and execute custom **HLSL** code directly within Shader Graph. Unlike Unreal Engine, Unity's documentation does not mention any performance loss associated with using the Custom Node for shader development. Additionally, Unity's Custom Node supports referencing external files or snippets, providing developers with added flexibility to manage and reuse shader code. This approach balances accessibility and customization, allowing developers to hook into Unity's lighting systems or extend functionality with minimal effort compared to pure code-based shaders. While Shader Graph lacks the low-level optimization flexibility of ShaderLab, its intuitive workflow, integration capabilities, and support for custom code make it a preferred choice for many modern Unity projects.

Godot

Godot takes a streamlined approach to code-based shaders with its own custom shader language. This language is designed to resemble **GLSL** and **HLSL**, making it approachable for developers familiar with these standards. However, Godot's shader system simplifies many tasks by automating common operations:

- **Vertex Attributes:** Default vertex attributes, such as position and normals, are automatically passed to the vertex shader, and their interpolated values are made available in the fragment shader.

50. Shadowriver, Writing Custom HLSL Shaders.

- **Built-in Varyings:** Developers can access interpolated values (e.g., UV coordinates) or modify them during processing. This system eliminates the need to manually define and manage varyings between shader stages.
- **Default Behavior:** If a developer leaves the vertex shader empty, Godot automatically calculates clip-space positions and passes through attributes like UVs and normals.

This combination of automation and flexibility allows developers to focus on the specific behavior they want to implement without worrying about boilerplate setup. Additionally, Godot's built-in shader editor provides immediate feedback, streamlining the debugging process and making shader development highly efficient.

Unlike Unity, Godot does not require an external tool for shader editing, and errors are displayed directly in the editor. Furthermore, the ability to access built-in passes, such as the light pass, enables developers to easily hook into Godot's existing lighting systems. By combining automation, flexibility, and real-time feedback, Godot offers an intuitive and efficient workflow for custom code based shaders.

Key Comparisons

- **Integration:** Unreal offers the most powerful rendering system but requires extensive knowledge to create custom code based shaders, especially when integrating with existing systems. Unity simplifies the workflow with ShaderLab but requires additional effort for deep integrations. Godot strikes a balance by automating most tasks while allowing manual overrides for custom behavior. Yet also has the least advanced existing rendering features.
- **Optimization:** Unity provides robust tools for targeting specific platforms and optimizing shaders with custom flags and models. Godot automates much of this, simplifying development at the cost of granular control. Unreal offers unmatched flexibility for optimization but at the cost of a steep learning curve.
- **Development Workflow:** Godot's built-in editor provides immediate feedback and access to all passes, making it the most developer-friendly environment for rapid shader iteration. Unity relies on external tools, while Unreal's code based workflows are close to none existing..

Comparison of Visual Node-Based Systems

The visual node-based systems for creating shaders in Unreal Engine, Unity, and Godot reflect their distinct approaches to balancing ease of use, flexibility, and integration with their respective rendering architectures.

Unreal Engine

Unreal Engine's Material Editor stands out as the most feature-rich among the three engines, offering an extensive library of nodes and tight integration with advanced rendering systems such as Lumen and Nanite. This integration allows developers to use these systems seamlessly, with the Material Editor managing complexities like lighting and geometry optimization behind the scenes. Adjusting material properties automatically hooks into these systems, ensuring optimal performance and visual quality.

Developers seeking more control can use Custom Expression nodes to write **HLSL** directly. While powerful, Epic Games advises caution when using Custom Expressions due to their inability to leverage optimizations like constant folding, which can lead to performance overheads. Despite this, the ability to write custom **HLSL** code and view the generated shader provides advanced users with significant flexibility, though the complexity of Unreal's rendering system can make this daunting for newcomers. Unreal's strong official documentation and vibrant community further enhance the learning experience, with numerous tutorials and resources available to guide developers in mastering the Material Editor.

Unity

Unity's Shader Graph has matured significantly, particularly for the **URP** and **HDRP**. Unity's node editor is well-rounded and highly capable, allowing developers to create sophisticated materials efficiently. The

Shader Graph provides a streamlined experience that accommodates developers of varying expertise, enabling them to create complex shaders without requiring in-depth knowledge of rendering pipelines.

Custom Nodes in Shader Graph allow developers to integrate **HLSL** code seamlessly. Unlike Unreal's Custom Expressions, Unity's Custom Nodes support referencing external files or snippets, enhancing flexibility. Additionally, Unity's documentation does not mention any significant performance trade-offs associated with Custom Nodes, making them a practical choice for developers who need advanced functionality.

Unity benefits from broad community support, with a wealth of tutorials and guides available. However, frequent updates to Shader Graph have sometimes led to compatibility issues between versions, making it necessary for developers to adapt older tutorials to newer versions. Nevertheless, Unity's active community and official resources mitigate these challenges, ensuring ample support for developers navigating Shader Graph's evolution.

Godot

Godot's visual shader editor takes a distinct approach, offering separate graphs for vertex, fragment, and light passes. This structure gives developers fine-grained control over the shader workflow. However, it also requires manual management of interpolated values between passes, adding complexity to shader development. While this setup can feel cumbersome for materials that utilize all three passes, it also provides greater control over the data flow, allowing developers to customize behavior precisely.

A unique advantage of Godot is that the generated shader code remains in Godot's proprietary shader language, which is concise and highly readable for users familiar with the engine. Additionally, developers can use Custom Nodes to write code directly in Godot's shader language. While this approach limits the ability to reference external files, it simplifies debugging and integration, especially given the built-in shader editor that provides immediate feedback during development.

Summary

Unreal Engine's Material Editor excels in complexity and integration, making it the go-to choice for high-fidelity projects, albeit with a steep learning curve for advanced customizations. Unity's Shader Graph strikes a balance between accessibility and functionality, offering flexibility for developers while maintaining a straightforward workflow. Despite occasional compatibility challenges with older tutorials, Unity benefits from broad community support that ensures a strong foundation for developers. Godot's visual shader editor prioritizes modularity and control, making it ideal for those who value direct manipulation of shader stages but potentially overwhelming for complex materials spanning multiple passes. Each system reflects its engine's philosophy, catering to different developer needs and expertise levels.

Discussion

The findings presented in this paper highlight key trends in shader programming, rendering pipelines, and engine adoption, with significant implications for developers choosing tools and workflows for game development.

Engine Adoption and API Trends

The analysis of released games over the past five years demonstrates that Unity and Unreal Engine dominate the market, driven by their mature tools, rendering pipelines, and broad platform support. Both engines primarily rely on modern graphics **API**s such as Vulkan, DirectX 12, and Metal to achieve high-performance rendering on PC, mobile, and macOS platforms.

Console platforms introduce additional considerations due to their unique **GPU** architectures and associated **API**s. On PlayStation, developers use the **GNM API** or the higher-level **Graphics Core Next Metal Extended (GNMX)** abstraction layer. Shaders for PlayStation are typically authored in **PSSL**,

which shares syntactic and structural similarities with **HLSL**. Similarly, on Xbox, games are built using DirectX-based **API**s, where shaders are written in an **HLSL** variant tailored to the Xbox **GPU**. These platform-specific shading languages can often be directly translated or compiled into HLSL, simplifying shader authoring and enabling efficient cross-platform development for AAA projects.

The widespread use of **HLSL** across modern engines and **API**s, including Vulkan (through **SPIR-V** compilation), reinforces its position as the shader language of choice for high-fidelity rendering. HLSL's flexibility and strong tooling support make it particularly well-suited for projects targeting both PC and console platforms.

While modern engines are moving towards Vulkan and Metal for improved performance, the role of OpenGL persists in specific contexts. OpenGL remains relevant for lightweight engines and educational purposes, where its simplicity and cross-platform compatibility provide an ideal foundation for learning rendering concepts. Engines like Godot still use OpenGL for their compatibility renderer, while frameworks such as Ren'Py, GameMaker, and RPG Maker continue to rely on OpenGL due to its ease of use and suitability for less demanding applications.

This distinction between modern **API**s for high-fidelity rendering and OpenGL for lightweight or educational projects highlights the importance of aligning **API** and shader language choices with project goals and platform requirements.

Implications of Visual Tools

The rise of visual shader tools like Unreal Engine's **Material Editor**, Unity's **Shader Graph**, and Godot's **Visual Shader Editor** has transformed shader development. These tools abstract much of the complexity of shader programming, enabling designers and non-specialized developers to create advanced effects without writing shader code.

However, the value of these tools is maximized when developers possess a foundational understanding of the rendering pipeline. Tasks such as performance optimization, troubleshooting, and effect customization do not necessarily require low-level shader authoring, but they do require an awareness of what happens behind the scenes. A clear understanding of vertex processing, fragment shading, and **GPU** resource management allows developers to make informed decisions, optimize their shaders, and fully leverage visual tools.

Balancing Fidelity and Accessibility

The findings highlight a fundamental trade-off between rendering fidelity and accessibility in engine selection.

Unreal Engine excels in AAA development, offering tools like Lumen and Nanite that push the boundaries of photorealism. However, the complexity of these tools, coupled with Unreal's advanced rendering pipelines, demands greater expertise and a steeper learning curve, at least for customizability.

Unity provides a balance between accessibility and advanced rendering features, with its **URP** catering to scalable projects and its **HDRP** enabling high-fidelity visuals. Unity's flexibility makes it ideal for projects targeting multiple platforms, from mobile to consoles.

Godot focuses on lightweight workflows and forward rendering, making it particularly effective for smaller teams, 2D projects, and less demanding 3D applications. While it lacks the advanced lighting capabilities of deferred rendering pipelines, its streamlined approach and visual shader tools provide accessible solutions that align with its goals.

While rendering workflows play a central role in engine selection, developers must also consider factors like platform support, built-in systems for gameplay programming, and tools for designers and artists. Engines like Unreal and Unity offer comprehensive ecosystems that cater to AAA and cross-platform projects, whereas Godot prioritizes simplicity and efficiency for smaller-scale development.

Future Considerations

Future research could expand on these findings by:

1. Benchmarking shader performance across visual tools and low-level code for advanced effects.
2. Analyzing the role of rendering pipelines in performance optimization for forward versus deferred lighting.
3. Investigating emerging engines and lightweight tools targeting VR, AR, and mobile development.

Conclusion

This paper provides a detailed analysis of shader languages, graphics **API**s, and shader workflows across Unity, Unreal Engine, and Godot, with an emphasis on engine adoption trends and practical implications for developers.

Key Findings

- **Engine Adoption**: Unity and Unreal Engine dominate the market, while Godot offers a strong alternative for lightweight projects.
- **API Trends**: Modern **API**s like Vulkan and Metal are increasingly replacing OpenGL in high-performance engines, though OpenGL remains relevant for lightweight applications and educational purposes.
- **Shader Workflows**: Visual tools simplify shader creation but require developers to maintain foundational knowledge of rendering pipelines for optimization and advanced customization.

Practical Implications

Developers must align engine choice with project requirements:
- **Unreal Engine** provides superior tools for photorealism and large-scale projects.
- **Unity** balances accessibility, flexibility, and platform scalability.
- **Godot** excels in lightweight workflows and accessibility, making it ideal for smaller teams.

Final Remarks

While visual shader tools have democratized shader development, mastering the fundamentals of shader programming remains critical. Modern engines and **API**s provide abstraction layers that simplify cross-platform development, but developers who understand the rendering pipeline at both high and low levels will be best equipped to balance performance and visual fidelity across diverse platforms.

Supplementary Material

The data extraction tool used in this study is publicly available in the following repository: https://dev.azure.com/EMSTEADania/IGDBGetter/_git/IGDBGamesGetter. This repository contains the source code for the C# program used to extract the dataset fields.
Additionally, all processed datasets, intermediate results, and analysis scripts are available at https://dev.azure.com/EMSTEADania/IGDBGetter/_git/IGDBDataAnalyzer. These resources ensure transparency and reproducibility of the study.

Appendix

References

Abramowicz, Kamil, and Przemysław Borczuk. 2024. Comparative Analysis of the Performance of Unity and Unreal Engine in 3D Games. *Journal of Computer Sciences Institute* 30:53–60.

Epic Games. 2021. Warning: OpenGL is No Longer Supported for Desktop Platforms. https://forums.unrealengine.com/t/warning-opengl-is-no-longer-supported-for-desktop-platforms/480923. Accessed: 2024-11-21.

Epic Games. Custom Material Expressions in Unreal Engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/custom-material-expressions-in-unreal-engine. Accessed: 2024-12-04.

Epic Games. Customizing Device Profiles and Scalability in Unreal Engine Projects for Android. https://dev.epicgames.com/documentation/en-us/unreal-engine/customizing-device-profiles-and-scalability-in-unreal-engine-projects-for-android. Accessed: 2024-12-04.

Epic Games. Essential Unreal Engine Material Concepts. https://dev.epicgames.com/documentation/en-us/unreal-engine/essential-unreal-engine-material-concepts. Accessed: 2024-12-04.

Epic Games. Nanite Virtualized Geometry in Unreal Engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/nanite-virtualized-geometry-in-unreal-engine. Accessed: 2024-12-04.

Epic Games. Render Hardware Interface (RHI) in Unreal Engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/render-hardware-interface-in-unreal-engine. Accessed: 2024-12-04.

Epic Games. Rendering Hardware Interface (RHI) Overview. https://dev.epicgames.com/documentation/en-us/unreal-engine/graphics-programming-overview-for-unreal-engine. Accessed: 2024-12-05.

Epic Games. Supported Features by Rendering Path for Desktop with Unreal Engine. https://dev.epicgames.com/documentation/en-us/unreal-engine/supported-features-by-rendering-path-for-desktop-with-unreal-engine. Accessed: 2024-12-05.

Epic Games. Unreal Engine Material Editor UI. https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-material-editor-ui. Accessed: 2024-12-04.

Galvan, Alain. 2021. A Review of Shader Languages. https://alain.xyz/blog/a-review-of-shader-languages. Accessed: 2024-11-21.

Godot Engine Developers. n.d.(a). Consoles — Godot Engine (stable) documentation in English. https://docs.godotengine.org/en/latest/tutorials/platform/consoles.html. Accessed: 2024-11-21.

Godot Engine Developers. n.d.(b). Rendering Drivers in Godot 4. https://docs.godotengine.org/en/stable/tutorials/rendering/rendering_drivers.html. Accessed: 2024-11-22.

Godot Engine Documentation Team. Internal Rendering Architecture. https://docs.godotengine.org/en/stable/contributing/development/core_and_modules/internal_rendering_architecture.html. Accessed: 2024-12-05.

Hussain, Faizan, Afzal Hussain, Haad Shakeel, Nasir Uddin, and Turab Latif Ghouri. 2020. Unity Game Development Engine: A Technical Survey. *University of Sindh Journal of Information and Communication Technology* 4 (2):1–10. Available at <https://www.researchgate.net/publication/348917348>.

Karis, Brian. 2013. Real Shading in Unreal Engine 4. Unpublished, *Epic Games,* Available at <https://www.epicgames.com/>.

Khronos Group. n.d. High-Level Shader Language Comparison. https://docs.vulkan.org/guide/latest/high_level_shader_language_comparison.html. Accessed: 2024-11-21.

Lee, HanSeong, SeungTaek Ryoo, and SangHyun Seo. 2019. A Comparative Study on the Structure and Implementation of Unity and Unreal Engine 4. *Korea Computer Graphics Society* 25 (4):17–24. https://doi.org/10.15701/kcgs.2019.25.4.17.

Ludwig, Heiko. 2016. Mapping between HLSL and GLSL. https://anteru.net/blog/2016/mapping-between-hlsl-and-glsl/. Accessed: 2024-11-21.

Ren'Py Developers. n.d. Ren'Py Visual Novel Engine. https://github.com/renpy/renpy. Accessed: 2024-11-22.

Sabir, Aqsa, Rahat Hussain, Akeem Pedro, and Lee Dongmin. 2024. Synthetic Data Generation with Unity 3D and Unreal Engine for Construction Hazard Scenarios: A Comparative Analysis. In Conference on Construction Hazard Scenarios. ResearchGate. Available at <https://www.researchgate.net/publication/382888381>.

Shadowriver. 2018. Writing Custom HLSL Shaders. Epic Games Developer Community Forums. Accessed: 2024-12-16. Available at <https://forums.unrealengine.com/t/writing-custom-hlsl-shaders/424362>.

Šmíd, Antonín. 2017. Comparison of Unity and Unreal Engine. Bachelor's Thesis. PhD diss., Czech Technical University in Prague, Faculty of Electrical Engineering. Available at <https://www.cvut.cz/>.

Stenson, Richard, and Chris Ho. 2014. PlayStation Shader Language for PlayStation 4. https://www.gdcvault.com/play/1019252/PlayStation-Shading-Language-for. Accessed: 2024-11-21.

Szabat, Bartłomiej, and Małgorzata Plechawska-Wójcik. 2023. Comparative Analysis of Selected Game Engines. *Journal of Computer Sciences Institute* 29:312–316.

Szafran, Kamil, and Małgorzata Plechawska-Wójcik. 2023. Impact of Changes in Graphics Setting on Performance in Selected Video Games. *Journal of Computer Sciences Institute* 28:291–295.

Szelug, Wojciech. 2022. Comparative Analysis of the Performance of the Flax Engine and Unity. *Journal of Computer Sciences Institute* 25:358–361.

Technologies, Unity. 2023. Unity's Future with Render Pipelines. YouTube Video. Accessed: 2024-11-22. Available at <https://www.youtube.com/watch?v=o9AGkB9nnkc>.

Unity Technologies. 2023. Unity OpenGL Deprecation and Removal for macOS and Windows. https://discussions.unity.com/t/opengl-deprecation-and-removal-for-macos-and-windows/888749. Accessed: 2024-11-21.

Unity Technologies. Choosing a Render Pipeline. https://docs.unity3d.com/Manual/choose-a-render-pipeline.html. Accessed: 2024-11-22.

Unity Technologies. GLSL Shader Programs. https://docs.unity3d.com/6000.0/Documentation/Manual/SL-GLSLShaderPrograms.html. Accessed: 2024-11-22.

Unity Technologies. n.d.(a). Graphics APIs on ios. https://docs.unity3d.com/6000.0/Documentation/Manual/android-requirements-and-compatibility.html. Accessed: 2024-11-22.

Unity Technologies. n.d.(b). Graphics APIs on ios. https://docs.unity3d.com/6000.0/Documentation/Manual/ios-requirements-and-compatibility.html. Accessed: 2024-11-22.

Unity Technologies. n.d.(c). Graphics APIs on Windows. https://docs.unity3d.com/Manual/GraphicsAPIs.html. Accessed: 2024-11-22.

Unity Technologies. Render Pipelines Feature Comparison. https://docs.unity3d.com/6000.0/Documentation/Manual/render-pipelines-feature-comparison.html. Accessed: 2024-11-22.

Unity Technologies. Writing Shaders for the Universal Render Pipeline. https://docs.unity3d.com/6000.0/Documentation/Manual/urp/writing-shaders-urp-basic-unlit-structure.html. Accessed: 2024-11-22.

Wikipedia Contributors. n.d. GameMaker Studio Graphics APIs. https://en.wikipedia.org/wiki/GameMaker. Accessed: 2024-11-22.

Acronyms

GPU Graphics Processing Unit

API Application Programming Interface

HLSL High-Level Shader Language

GLSL OpenGL Shading Language

MSL Metal Shading Language

PSSL PlayStation Shader Language

SPIR-V Standard Portable Intermediate Representation for Vulkan

URP Universal Render Pipeline

HDRP High Definition Render Pipeline

RHI Rendering Hardware Interface

GNM Graphics Core Next Metal

GNMX Graphics Core Next Metal Extended

SM Shader Model

IGDB International Game Database

SRP Scriptable Render Pipeline

CG C for Graphics

Authors

Emil Stephansen is a assistant Lecturer at Erhvervsakademi Dania, specializing in game development, shader programming, and network programming. He holds a Master's degree in Computer Science from Aarhus university and conducts research on rendering technologies, graphics **API**s, and shader development. Contact: EMST@eadania.dk.

Acknowledgements

This draft is currently under internal review.

Key Words

Game Engines, Shader Programming, Graphics APIs, Unity, Unreal Engine, Godot, Vulkan, DirectX 12, Metal, OpenGL, HLSL, SPIR-V, Forward Rendering, Deferred Rendering, Visual Shader Editors, Material Editor, Shader Graph

TABLE A1. ***GPU API*** *used on different platforms*

Engine	Platform	**API**	Notes
Unity	PC (Microsoft Windows)	DirectX12, DirectX11, Vulkan, OpenGL, OpenGLES3	OpenGL is deprecated see[51]
Unity	Mac	Metal	
Unity	Linux	OpenGL, Vulkan[52]	
Unity	PlayStation 5	**GNM**	
Unity	PlayStation 4	**GNM**[53]	**GNM** assumed based on PS architecture; official documentation unavailable.
Unity	Xbox Series X|S	DirectX11, DirectX12	
Unity	Xbox One	DirectX11, DirectX12	
Unity	Nintendo Switch	NVN	
Unity	Android	Vulkan, OpenGLES[54]	
Unity	iOS	Metal[55]	
Unity	Web	WebGL	
Unreal	PC (Microsoft Windows)	DirectX12, DirectX11, Vulkan, OpenGL	OpenGL is deprecated see[56]
Unreal	Mac	Metal	
Unreal	Linux	Vulkan, OpenGL	OpenGL is deprecated see[57]
Unreal	PlayStation 5	**GNM**	
Unreal	PlayStation 4	**GNM**	
Unreal	Xbox Series X|S	DirectX11, DirectX12	
Unreal	Xbox One	DirectX11, DirectX12	
Unreal	Nintendo Switch	NVN	
Unreal	Android	Vulkan, OpenGLES	
Unreal	iOS	Metal	
Unreal	Web	WebGL	
Godot[58]	PC (Microsoft Windows)	DirectX12, Vulkan, OpenGLES	
Godot	Mac	Metal, OpenGLES	Uses Vulkan transpiled to Metal via MoltenVK.

Continued on next page

51. Unity Technologies, Unity OpenGL Deprecation and Removal for macOS and Windows.
52. Unity Technologies, Graphics APIs on Windows.
53. Stenson, Richard, and Chris Ho, PlayStation Shader Language for PlayStation 4.
54. Unity Technologies, Graphics APIs on ios.
55. Unity Technologies, Graphics APIs on ios.
56. Epic Games, Warning: OpenGL is No Longer Supported for Desktop Platforms.
57. Epic Games, Warning: OpenGL is No Longer Supported for Desktop Platforms.
58. Godot Engine Developers, Rendering Drivers in Godot 4.

Engine	Platform	**API**	Notes
Godot	Linux	Vulkan, OpenGLES	
Godot	PlayStation 5	N/A	
Godot	PlayStation 4	N/A	
Godot	Xbox Series X|S	N/A	
Godot	Xbox One	N/A	
Godot	Nintendo Switch	N/A	
Godot	Android	Vulkan, OpenGLES	
Godot	iOS	Metal, OpenGLES	Uses Vulkan transpiled to Metal via MoltenVK; OpenGL ES is available as a fallback for legacy devices.
Godot	Web	WebGL	
GameMaker[59]	PC (Microsoft Windows)	DirectX11	
GameMaker	Mac	OpenGL	
GameMaker	Linux	OpenGL	
GameMaker	PlayStation 5	**GNM**	
GameMaker	PlayStation 4	**GNM**	
GameMaker	Xbox Series X|S	DirectX11	
GameMaker	Xbox One	DirectX11	
GameMaker	Nintendo Switch	OpenGLES	
GameMaker	Android	OpenGLES	
GameMaker	iOS	OpenGLES	
GameMaker	Web	WebGL	
RenPy[60]	PC (Microsoft Windows)	OpenGL	
RenPy	Mac	OpenGL	
RenPy	Linux	OpenGL	
RenPy	PlayStation 5	N/A	
RenPy	PlayStation 4	N/A	
RenPy	Xbox Series X|S	N/A	
RenPy	Xbox One	N/A	
RenPy	Nintendo Switch	N/A	
RenPy	Android	OpenGLES	
RenPy	iOS	OpenGLES	
RenPy	Web	WebGL	

Date received: N/A; Date accepted: N/A.

59. Wikipedia Contributors, GameMaker Studio Graphics APIs.

60. The use of OpenGL and OpenGL ES in Ren'Py is not directly documented but is observable in the engine's open-source code on GitHub: Ren'Py Developers, n.d.