



MASTERARBEIT | MASTER'S THESIS

Titel | Title

Exploring WebGPU as a Foundation for a Game Engine and
Teaching Tool

verfasst von | submitted by

Kalian Danzer BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 921

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Informatik

Betreut von | Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Acknowledgements

I would like to express my sincere gratitude to Prof. Helmut Hlavacs for supervising this project and for his support throughout the entire research and development process. His expertise and feedback in graphics programming and the design of educational game engines were crucial to the development of this work.

Special thanks go to all friends and students who participated in the evaluation study. Their time and honest feedback were essential for assessing the educational effectiveness of the Vienna WebGPU Engine and its tutorials. Without their contributions and the feedback loop during development, this project would not have been possible.

A big thank you goes to Élie Michel for creating the tutorial series “LearnWebGPU - WebGPU C++ Guide” which provided an excellent foundation for understanding WebGPU concepts, and whose resources were a great help in making the project run on various platforms.

In closing, I would like to express my sincere gratitude to my family for their support throughout my studies, which allowed me to focus entirely on my degree. I would also like to thank my employer for the flexibility in working hours and location that made it possible to pursue this degree alongside professional responsibilities.

Abstract

WebGPU is a modern, low-level graphics API that provides direct access to GPU resources across web browsers and native applications. It represents a promising foundation for both web and desktop graphics systems, offering explicit control over GPU operations similar to Vulkan, Metal, and DirectX 12. By exposing GPU pipelines, memory, and shader management directly to developers, WebGPU enables high performance and parallelism, but also introduces additional complexity compared to higher-level APIs. This thesis investigates three questions: What is the current state of WebGPU as a graphics API? Can a fully functional game engine be built on top of it? Can such an engine be designed to teach WebGPU concepts effectively?

To address these questions, a modular **WebGPU-based game engine**, the **Vienna WebGPU Engine**, was developed, implementing fundamental systems such as physically-based rendering, hierarchical scene organization, resource management, multi-camera support, and shadow mapping. The engine is designed with functionality and clarity in mind, making its subsystems and data flow accessible for educational purposes. Using this engine, a **mini-course** with three tutorials was created to teach WebGPU through guided, hands-on exercises, allowing participants to explore rendering, materials, lighting, and GPU resource management through tasks of increasing complexity.

The educational approach was evaluated through a three-stage process: a **pre-questionnaire** assessed baseline confidence and prior experience using a 4-point Likert scale, participants completed the **guided tutorials** independently while being observed, and a **post-questionnaire** measured learning gains and perceived effectiveness using the same scale supplemented by evaluation items on helpfulness and willingness to recommend the materials.

This work was conducted within the EDEN research group (Education, Didactics and Entertainment Computing) at the University of Vienna, where established courses such as *Gaming Technologies* and *Cloud Gaming* currently employ the Vienna Vulkan Engine and Vienna Physics Engine as teaching frameworks for teaching graphics programming and game development. The Vienna WebGPU Engine explores WebGPU as a potential future alternative or extension to these existing teaching tools, examining whether WebGPU's design characteristics and cross-platform capabilities offer pedagogical advantages that could complement or enhance current course materials. This work shows that WebGPU can support a fully functional game engine and that such an engine provides a viable platform for hands-on, educational experiences in modern graphics programming that could be integrated into university curricula as structured exercises, bonus projects, or foundations for bachelor's and master's theses.

Kurzfassung

WebGPU ist eine moderne, low-level Grafik-API, die direkten Zugriff auf GPU-Ressourcen sowohl in Webbrowsern als auch in nativen Anwendungen ermöglicht. Sie stellt eine vielversprechende Grundlage für Web- und Desktop-Grafiksysteme dar und bietet explizite Kontrolle über GPU-Operationen ähnlich wie Vulkan, Metal und DirectX 12. Durch die direkte Bereitstellung von GPU-Pipelines, Speicherverwaltung und Shader-Management an Entwickler ermöglicht WebGPU hohe Leistung und Parallelismus, führt aber auch zu zusätzlicher Komplexität im Vergleich zu höheren APIs. Diese Arbeit untersucht drei Fragen: Was ist der aktuelle Stand von WebGPU als Grafik-API? Kann eine voll funktionsfähige Game-Engine darauf aufgebaut werden? Und kann eine solche Engine so gestaltet werden, dass sie WebGPU-Konzepte effektiv vermittelt?

Um diese Fragen zu beantworten, wurde eine modulare **WebGPU-basierte Game-Engine**, die **Vienna WebGPU Engine**, entwickelt, die grundlegende Systeme wie physikalisch basiertes Rendering, hierarchische Szenenorganisation, Ressourcenverwaltung, Multi-Kamera-Unterstützung und Shadow-Mapping implementiert. Die Engine ist sowohl auf Funktionalität als auch auf Verständlichkeit ausgelegt, wodurch ihre Subsysteme und Datenflüsse für Lehrzwecke zugänglich werden. Mit dieser Engine wurde ein **Mini-Kurs** mit drei Tutorials erstellt, der WebGPU durch angeleitete, praxisnahe Übungen vermittelt und es den Teilnehmern ermöglicht, Rendering, Materialien, Beleuchtung und GPU-Ressourcenverwaltung durch Aufgaben steigender Komplexität zu erkunden.

Der didaktische Ansatz wurde durch einen dreistufigen Prozess evaluiert: Ein **Vorher-Fragebogen** erfasste das Grundvertrauen und die Vorkenntnisse mittels einer 4-Punkte-Likert-Skala, die Teilnehmer bearbeiteten die **angeleiteten Tutorials** selbstständig unter Beobachtung, und ein **Nachher-Fragebogen** maß Lernerfolge und wahrgenommene Effektivität mit derselben Skala, ergänzt durch Evaluationsitems zu Hilfestellung und Bereitschaft, die Materialien weiterzuempfehlen.

Diese Arbeit wurde innerhalb der EDEN-Forschungsgruppe (Education, Didactics and Entertainment Computing) an der Universität Wien durchgeführt, wo etablierte Lehrveranstaltungen wie *Gaming Technologies* und *Cloud Gaming* derzeit die Vienna Vulkan Engine und Vienna Physics Engine als didaktische Frameworks zur Vermittlung von Grafikprogrammierung und Spieleentwicklung einsetzen. Die Vienna WebGPU Engine untersucht WebGPU als potenzielle zukünftige Alternative oder Erweiterung zu diesen bestehenden Lehrmitteln und prüft, ob WebGPUs Designmerkmale und plattformübergreifende Fähigkeiten pädagogische Vorteile bieten, die aktuelle Kursmaterialien ergänzen oder verbessern könnten. Diese Arbeit zeigt, dass WebGPU nicht nur eine voll funktionsfähige Game-Engine unterstützen kann, sondern auch eine praktikable Plattform für praxisnahe, didaktische Erfahrungen in moderner Grafikprogrammierung bietet, die in universitäre Curricula als strukturierte Übungen, Bonusprojekte oder als Grundlage für Bachelor- und Masterarbeiten integriert werden könnte.

Contents

Acknowledgements	i
Abstract	ii
Kurzfassung	iii
List of Tables	vi
List of Figures	vii
Listings	viii
1. Introduction	1
2. Related Work	4
3. Technical Foundations	15
3.1. WebGPU Architecture, Programming Model, and Resource Management . . .	15
3.2. Physically-Based Rendering Theory	21
3.3. Scene Graph and Entity Organization	23
3.4. Design Patterns for Graphics Systems	24
4. Vienna WebGPU Engine: Educational Rendering Engine Architecture	27
4.1. WebGPU Platform State and Implementation Landscape	28
4.2. Layered Architecture, Handle System, and Scene Graph	31
4.3. Shader Reflection, Pipeline Automation, and Multi-Pass PBR Rendering . .	38
4.4. Tutorial Series and Progressive Learning Design	46
4.5. Implementation Summary and Cross-Platform Considerations	49
5. Evaluation	51
5.1. RQ1: WebGPU Ecosystem Assessment	51
5.2. RQ2 and H1: Implementation Complexity Comparison	54
5.3. RQ3: Educational Effectiveness User Study	55
5.4. Discussion	64
6. Conclusion	66
Bibliography	68
A. Appendix	74
A.1. Reproducibility Artifacts for Evaluation	74

Contents

A.2. Responsibility Delegation for Progressive Learning	78
A.3. AI Tool Usage Documentation	79

List of Tables

- 4.1. Comparison of WebGPU implementations considered for the engine 29
- 5.1. RQ1 evidence overview: WebGPU ecosystem maturity for educational use
(early 2025 to February 2026). 52
- 5.2. RQ1 qualitative cross-API positioning for educational use. 53
- 5.3. Per-file lizard metrics for the H1 canonical baseline pair. 54
- 5.4. Questionnaire items (Q01 to Q33) with availability and topic mapping. . . . 57
- 5.5. Participant profile and tutorial completion overview ($n = 15$). 57
- 5.6. Internal consistency of questionnaire blocks (Cronbach’s α). 59
- 5.7. Composite pre/post statistics for Q01 to Q25 ($n = 15$). 59
- 5.8. H2 inferential summary for participant-level Q01 to Q25 gains ($n = 15$). . . 59
- 5.9. H3 item-level agreement tests (one-sided exact binomial, Holm-adjusted). . 63
- A.1. Evaluation data artifacts for reproducibility (see repository links above) . . 75
- A.2. Overview of AI tools used during thesis creation 79

List of Figures

4.1.	Multi-pass rendering pipeline with per-camera execution (red box) and data dependencies (blue arrows).	44
4.2.	Main rendering scene illustrating the engine’s practical visual output. . . .	45
4.3.	Example of a tutorial step showing the code block, contextual explanation, and integration points.	47
4.4.	Tutorial progression shown as a 2x2 visual sequence from baseline scene setup to post-processing integration across tutorial milestones.	48
5.1.	H1 complexity metrics for the canonical baseline pair.	54
5.2.	Combined topic-level H2 evidence.	60
5.3.	Participant-level heatmap of Q01 to Q25 deltas (POST – PRE). Greener colors indicate larger positive gains. More reddish color tones indicate decreases.	61
5.4.	Mean score increase per item across Q01 to Q25. High-growth items align with core tutorial constructs.	62
5.5.	Participant-level ranking by mean gain across Q01 to Q25.	62
5.6.	Combined H3 item view: top panel shows agreement rates (responses ≥ 3), bottom panel shows stacked Likert distributions for Q26 to Q33 (post-only block).	64

Listings

- 4.1. Handle System with Resolver Pattern 35
- 4.2. Manual Bind Group Layout Creation for One Uniform Buffer 39
- 4.3. ShaderFactory Builder Pattern for Shadow Pass Shader 40
- A.1. H1 LOC and API-call counting script 76

1. Introduction

The field of graphics programming is characterized by a constant tension between performance and accessibility. Modern graphics applications require explicit control of GPU resources to achieve optimal performance. However, the complexity of low-level APIs such as Vulkan and DirectX 12 poses a considerable obstacle for students, educators, and independent developers learning graphics programming [Gal21]. These APIs unlock the full potential of modern graphics hardware through detailed resource management, explicit synchronization primitives, and finely tuned pipeline configuration. However, developers must grasp hundreds of concepts before they can render even a simple triangle. This gap in complexity has created a need for teaching frameworks that link high-level abstractions and low-level GPU programming [Gre18], allowing learners to understand modern graphics architecture without being overwhelmed by implementation details. The goal is to design systems that retain the educational benefits of explicit APIs while providing a structure that lowers the learning barrier and enables the gradual learning of complex concepts.

WebGPU is a promising candidate for addressing this educational challenge. As a modern, cross-platform graphics API developed by the W3C GPU for the Web Community Group and standardized by the W3C GPU for the Web Working Group [W3C23b], WebGPU draws on the design principles of Vulkan, Metal, and DirectX 12 [Khr16, App14, Mic15] and targets both web browsers and native applications. Unlike its predecessor WebGL, which abstracted many GPU details through an OpenGL-like interface, WebGPU exposes explicit resource management, command encoding, and pipeline configuration [CBH⁺24] that correspond to modern native APIs. This explicit design philosophy makes WebGPU educationally valuable for teaching contemporary GPU programming concepts, as students learn patterns that translate directly to professional graphics development [Ken22]. However, the relative novelty of WebGPU raises important questions about its maturity, ecosystem support, and suitability as a foundation for educational tools. While the W3C specification remains at Candidate Recommendation stage as of February 2026, WebGPU has achieved stable support across all major desktop browsers, and the core `webgpu.h` C API header has been declared stable as of Chrome 141 [Chr25c]. This thesis investigates WebGPU’s current state as a mature alternative to established graphics APIs, examining whether it has achieved the stability and ecosystem support necessary for educational adoption. The implementation and tutorial material in this thesis were developed from early 2025 to February 2026 while keeping a stable WebGPU baseline from that period to avoid disruptions from rapid API and toolchain changes.

The motivation for this work originates from the observation that existing graphics education typically follows one of two unsatisfactory paths. Traditional approaches teach advanced engines such as Unity or Unreal [Uni22, God23], which hide GPU details behind extensive abstractions and make it difficult for students to understand the underlying rendering architecture. While students can quickly achieve impressive visual results, they lack knowledge about how graphics pipelines actually work, how memory is managed on

1. Introduction

the GPU, or how rendering performance can be optimized [HZ20]. Conversely, low-level APIs such as Vulkan present a significant barrier for inexperienced students, requiring extensive boilerplate code, such that canonical onboarding tutorials can approach 1000 lines of code before any meaningful rendering takes place [DPH16, UKW23]. The confirmatory canonical baseline used later in this thesis is narrower and reports around 700 NLOC for Vulkan in the selected hello-triangle reference (Section 5.2). While teaching Vulkan has been shown to be viable in structured academic settings [UKW23], the steep initial complexity motivates the need for more accessible alternatives. A middle ground is needed: a teaching framework based on modern API principles that teaches important GPU concepts through clean abstractions without introducing unnecessary complexity. The design philosophy of WebGPU fits well with this goal, but there is no comprehensive assessment of whether a production-quality game engine can be built on WebGPU and whether such an engine can effectively serve as a teaching tool. This work fills this gap by designing, implementing, and evaluating the *Vienna WebGPU Engine*. This research addresses three primary questions:

RQ1: What is the current state of WebGPU as a graphics API compared to established low-level alternatives?

RQ2: Is it feasible to build a fully functional rendering engine using WebGPU?

RQ3: Can such an engine effectively teach modern GPU programming concepts to students?

RQ3 requires particular attention to the design of the engine and its accompanying tutorials. The engine was designed with consistent use of established patterns such as the Factory Pattern, extensive documentation, and a tutorial structure that introduces GPU concepts incrementally, with the goal of making modern GPU programming accessible to students encountering WebGPU for the first time.

RQ1 and RQ2 provide context and feasibility evidence, while RQ3 is the primary research focus. This thesis proposes three testable hypotheses addressing these questions:

H1 (addressing RQ2): Implementing equivalent rendering tasks requires significantly fewer lines of code and fewer distinct API calls (used as an operational proxy for API concepts) in WebGPU than in Vulkan [Mic22, Ove23], demonstrating WebGPU’s accessibility advantages.

H2 (addressing RQ3): Students report a statistically significant positive pre/post change in self-assessed GPU programming confidence after completing the Vienna WebGPU Engine tutorials. The primary confirmatory test is a one-sided paired Wilcoxon signed-rank test on Q01 to Q25 ($H_1 : \text{POST} > \text{PRE}$, $\alpha = 0.05$), while effect magnitude is reported with rank-biserial correlation and Cohen’s d_z , and uncertainty of the mean paired difference is reported using a 95% confidence interval from the paired t -test framework.

H3 (addressing RQ3): Students perceive the Vienna WebGPU Engine and its accompanying tutorials as an effective and approachable tool for learning modern GPU programming concepts. For Q26 to Q33, agreement is operationalized

1. Introduction

as responses 3 – 4 (Agree/Strongly Agree) and tested against $p_0 = 0.5$ with a one-sided exact binomial test. Effect strength is interpreted using observed agreement proportions and exact 95% confidence intervals.

This work was conducted within the EDEN research group (Education, Didactics and Entertainment Computing) at the University of Vienna, where graphics teaching already relies on hands-on engine-oriented coursework and coding-focused project formats [Hla24b, Hla24a]. The Vienna WebGPU Engine follows this pedagogical direction while targeting WebGPU as a modern cross-platform API. Detailed educational and architectural context is expanded in [Chapter 3](#) and [Chapter 4](#).

This thesis contributes three main findings at the intersection of graphics programming and computer science education. First, the implementation demonstrates that a fully functional and teaching-oriented WebGPU rendering engine is feasible, including PBR with Cook-Torrance BRDF [CT82, Kar13], multi-light shadowed rendering [ZSSL06], normal mapping, multi-camera support, a Unity-inspired scene graph, type-safe resource handles [Gra18], and a factory-centered architecture [GHJV94]. Second, the ecosystem analysis shows that WebGPU reached practical teaching maturity by early 2026, while backend fragmentation and toolchain drift still require explicit environment planning [DPU⁺22, Ust24, KWW23, JWB⁺25, RGM⁺25]. Third, the empirical evaluation provides evidence that WebGPU can reduce introductory implementation overhead in the selected canonical baseline and that the tutorial sequence is associated with positive self-assessed learning outcomes [Mic22, Ove23, UKW23]. The engine source code and documentation are available at <https://github.com/hlavacs/Vienna-WebGPU-Engine>.

A note on AI usage: AI-assisted tools were used for language refinement, structural polishing, code documentation, and locating technical references. Technical decisions, architecture, implementation, research design, and statistical evaluation were performed by the author. Detailed documentation of AI tool usage, including specific purposes and affected sections, is provided in [Appendix A.3](#).

The thesis is organized as follows. [Chapter 2](#) surveys GPU programming education, graphics API comparisons, and game engine design. [Chapter 3](#) covers necessary technical background: WebGPU architecture, GPU programming concepts, and design patterns. [Chapter 4](#) details the Vienna WebGPU Engine’s architecture, implementation, and capabilities. [Chapter 5](#) presents performance analysis and user study results. [Chapter 6](#) summarizes findings and proposes future directions.

2. Related Work

The evolution of GPU computing has fundamentally transformed computer graphics and parallel computation. Owens et al. [OHL⁺08] provided a comprehensive survey of general-purpose GPU computing, examining both hardware capabilities and programming paradigms across scientific and graphics workloads. They reported substantial performance improvements for parallel computation tasks while identifying challenges in handling irregular data patterns and the need for higher-level abstraction layers. Their work established a foundational understanding of GPU architectures that continues to inform modern graphics API design.

Luebke et al. [LHK⁺04] provided an early comprehensive introduction to general-purpose computing on graphics hardware (GPGPU), covering basic computational building blocks such as linear algebra and database queries and showing that GPU acceleration can outperform CPU performance by many times. The authors found that despite this potential, there were significant obstacles for developers due to novel and unusual programming, limited environments, and largely unknown hardware architectures. While this work laid important foundations for understanding GPU programmability, the focus remained on general computation rather than educating on structured graphics programming, a gap that motivates the development of accessible graphics-specific learning tools.

The transition to explicit low-level graphics APIs marked a paradigm shift in GPU programming. The Vulkan specification from the Khronos Group [Khr16] defines explicit low-level GPU control through detailed resource models and command buffer abstractions. It enables fine-tuned performance optimization and direct hardware access that were previously unavailable in higher-level APIs. The inherent complexity of the specification poses a hurdle for inexperienced programmers and educational use, motivating research into abstraction layers and teaching methods.

Microsoft’s DirectX 12 documentation [Mic15] describes in detail the programming model for Windows GPU applications, with a focus on command buffer and pipeline state object management. Performance improvements over DirectX 11 stem from reduced driver overhead and explicit multithreading support. The steep learning curve and platform-specific nature limit its more widespread use in education, especially for cross-platform or web-based teaching.

Apple’s Metal API [App14] offers GPU access optimized for Apple hardware and provides documentation covering shader programming and resource management techniques. Metal achieves high performance on iOS and macOS through tight hardware-software integration and low-level control. Platform exclusivity restricts educational adoption to institutions with Apple hardware, creating accessibility barriers compared to cross-platform and web-based alternatives for teaching modern GPU programming.

Galvan [Gal21] provides a technical comparison of modern graphics APIs, including Vulkan, DirectX 11/12, Metal, OpenGL, and WebGPU, listing structural differences while highlighting their fundamental similarities in GPU hardware sharing. The comparison

2. Related Work

highlights API-specific data structures such as device initialization, command buffers, pipelines, and synchronization primitives, revealing that WebGPU omits certain concepts found in other APIs, such as fences. This structural overview illustrates how modern APIs converge in their design philosophy while differing in specific implementation details, and provides valuable information for understanding WebGPU’s position within the graphics API landscape.

Unterguggenberger et al. [UKW23] documented experiences teaching Vulkan alongside OpenGL in both introductory and advanced university graphics courses, addressing the challenge of transitioning from the established OpenGL-based curriculum to modern low-level APIs. Through survey-based evaluation, they found that abstraction frameworks were important for managing workload: using Vulkan Launchpad to hide complex concepts for beginners while employing Auto-Vk for advanced courses to preserve fundamental understanding. Their results showed that C/C++ proficiency posed greater challenges than the graphics API itself for introductory students, while synchronization concepts proved most difficult in advanced courses, leading to their recommendation that teaching low-level APIs from the beginning better prepares students for modern GPU programming than later transitions from higher-level APIs. Student perception surveys revealed that while OpenGL students remained indecisive about which API would be more valuable for future studies and industry work, Vulkan students showed significantly higher confidence in the relevance and industry applicability of their acquired skills.

The W3C WebGPU specification [W3C23b] defines a modern graphics and computing API that makes GPU capabilities available via JavaScript for web environments, incorporating design insights from Vulkan, Metal, and DirectX 12 while taking into account web-specific security constraints. The specification introduces explicit resource management through bind groups, pipeline layouts, and command buffers, along with strict validation requirements to prevent CPU- and GPU-based undefined behavior that could compromise browser safety. Challenges in implementation include ensuring cross-browser consistency across different native backends, defining performance characteristics that match native APIs, and developing training materials that address the inherent complexity of the API for developers transitioning from WebGL. As of early 2026, the specification is still at the Candidate Recommendation stage, and full standardization is pending.

The W3C WGSL specification [W3C23a] introduces WebGPU Shading Language, a type-safe shader language with strict validation rules that enforce mandatory boundary checks and explicit memory layout specifications, thereby reducing common undefined behaviors. The stricter type system and explicit syntax ensure more consistent and predictable shader behavior across different GPU hardware and drivers, but at the cost of increased verbosity compared to established shading languages. However, WGSL is still novel and has limited tools, debugging support, and educational resources, so its adoption in academic graphics programming courses is still open.

Kenwright [Ken22] presented introductory WebGPU course materials at ACM SIGGRAPH 2022, in which he introduced the API using JavaScript examples and discussed various basic concepts such as adapter and device initialization, the structure of the render pipeline, and buffer management. The materials also include more detailed usage examples

2. Related Work

and primarily give developers familiar with web technologies an early overview of WebGPU. However, the examples date from a time before significant API changes and use JavaScript instead of a system programming language, which limits their applicability for teaching low-level GPU programming concepts in a C++ educational context.

Michel [Mic22] has developed “LearnWebGPU: WebGPU C++ Guide”, a comprehensive online tutorial series that walks through the implementation of WebGPU with wgpu-native step by step, from window creation to loading and rendering 3D meshes, while providing tools to support all major WebGPU backends such as wgpu-native, Dawn, and Emscripten. The tutorials progress step by step, using working code at each step, and cover core concepts such as buffers, textures, uniforms, and basic lighting, while several advanced sections such as PBR, scene graphs, and shadow mapping are not (yet) covered. However, the materials require extensive prior knowledge of graphics programming, focus more on setup and individual API concepts than on a complete engine architecture, and do not cover higher-level topics such as shadow mapping, render passes, shaders, or structured engine design patterns.

The WebGPU Samples project [Web23] is an official collection of browser-based examples maintained by the WebGPU community, which illustrates the WebGPU API through interactive demos ranging from basic rendering to more advanced GPU techniques. The samples serve as practical reference implementations for developers exploring WebGPU features in a JavaScript and TypeScript environment. However, the collection provides no structured learning progression or explanatory documentation, and being browser-based and written in TypeScript, it does not address native C++ GPU programming or engine-level architectural concerns.

Google’s Dawn [Goo26] is the native WebGPU implementation powering Chrome and its derivatives, written in C++ and mapping WebGPU calls to Direct3D 12, Metal, and Vulkan depending on the target platform. As the reference implementation used in the world’s most widely deployed browser, Dawn serves as the canonical baseline for WebGPU behavior and is also available as a standalone library for native C++ applications. While Dawn is a foundational infrastructure component rather than an educational tool, its availability as a standalone library enables native engine development on top of WebGPU outside the browser context.

Chickerur et al. [CBH⁺24] conducted systematic performance comparisons between WebGL and WebGPU on various visualization workloads to explore how user experience could be improved in Web 3.0 through WebGPU. They found significant performance advantages for WebGPU, but identified cross-browser consistency of implementation as a continuing challenge, which is to be expected at the time of the study in 2024. The study provided empirical data supporting the use of WebGPU for performance-critical applications and showed that switching to WebGPU brings a meaningful benefit.

In Palmér’s dissertation [Pal24], the performance differences between WebGPU and WebGL were analyzed specifically for 2D particle systems, examining GPU time and frame rates on two hardware configurations, revealing significant performance improvements from WebGPU on both high-end and low-end GPUs. However, it was noted that mid-range and mobile devices were not included in the study, and the difficulties posed by

2. Related Work

the lack of existing preliminary work and the unfinished nature of WebGPU were also discussed. The paper highlights that WGSL shaders have the advantage of compute shaders over GLSL shaders, which is a major advantage in modern engines, for example for particle systems, as many calculations can take place simultaneously on the GPU, making this work particularly interesting.

Kaltschmid [Kal24] provided an even more comprehensive evaluation of WebGPU as a general computing platform beyond graphics rendering, examining the capabilities of compute shaders, memory management, and performance characteristics for scientific computing tasks. Although the focus is more on computational tasks than graphics, the results are interesting for engine development, as compute shaders are an integral part of modern rendering pipelines and are used for tasks such as post-processing, particle simulation, and procedural geometry generation. The study leaves room for further investigation into how WebGPU’s compute capabilities can be integrated into a structured graphics engine architecture designed for use in education.

Sengupta et al. [SWV⁺25] tested WebGPU and WebGL for general GPU computations in browser environments and compared the performance of six combinations of execution environments (JavaScript and WebAssembly, each without GPU acceleration or with WebGL or WebGPU) for different input sizes. Their results show that GPU-based APIs only outperform CPU execution for large inputs due to overhead for setup and synchronization, and that WebGPU outperforms WebGL for large inputs due to optimized thread utilization, among other factors. This comparison is useful because WebGL is the dominant GPU API for web-based graphics, and many developers switching to WebGPU come from the WebGL space, providing a useful context for understanding where WebGPU offers real advantages over its predecessor.

Bi et al. [BMH⁺24] introduced FusionRender, a WebGPU-based graphics rendering framework that addresses the performance gap between current WebGPU and WebGL implementations by reducing redundant CPU-GPU communication through draw call merging, assigning unique signatures to objects and grouping them to consolidate rendering calls wherever possible. Their evaluation measured clear rendering performance improvements over existing WebGPU baselines and showed that naive WebGPU adoption without API-specific optimization does not automatically outperform WebGL. This is relevant to engine design as it highlights that efficient use of WebGPU requires architectural decisions around draw call batching and CPU-GPU synchronization, aspects that should be communicated to students learning GPU programming.

Dyken et al. [DPU⁺22] developed GraphWaGu, the first WebGPU-based system for graph visualization. It utilizes WebGPU compute shaders to implement parallel layout algorithms for large graphs directly in the browser, surpassing existing web-based visualization libraries like D3.js and NetV.js in terms of performance and scalability. The work demonstrates that the computational power of WebGPU extends far beyond traditional graphics rendering, enabling GPU-accelerated algorithms that were previously impractical in browser environments. This is relevant as an early practical validation of WebGPU’s suitability as a high-performance GPU platform and supports the development of more complex WebGPU-based systems. However, the focus remains on a specific use case of

2. Related Work

visualization rather than general engine architecture or educational applications.

Usta [Ust24] investigated WebGPU in the context of 3D WebGIS applications and conducted a direct performance comparison between WebGL and WebGPU when rendering 3D city models in the browser without using third-party libraries. The results show that WebGPU consistently outperforms WebGL at all tested dataset sizes, which is attributed to WebGPU’s low-level API design, explicit resource management, and native support for parallel processing and multithreading. The study also highlights that WebGPU has a higher verbosity compared to WebGL, as developers must manually manage tasks such as depth buffering, anti-aliasing, and canvas management. This underlines the need for higher-level abstractions and structured engine architectures.

Jiwad et al. [JWB⁺25] developed FlockJS, combining WebGPU rendering with WebRTC-based peer-to-peer networking, enabling development of multiplayer gameplay through an API that abstracts low-level details. Their evaluation shows that FlockJS sustains interactive frame rates with low latency in hybrid network topologies and feedback from student developers confirms the API’s clarity and ease of integration. The project indicates that abstracting WebGPU is a viable way to lower the barrier for educational use.

Rodríguez et al. [RGM⁺25] presented wgpuEngine, an open-source cross-platform C++ graphics engine built on WebGPU, supporting both desktop and web through Dawn and Emscripten with additional support for OpenXR. The engine achieves consistent feature parity across Windows, Linux, macOS, and web browsers from a single WebGPU-based rendering backend and is validated through real-world deployments. This work is closely related as it pursues a similar goal of building a structured C++ engine on top of WebGPU, though its focus lies on cross-platform XR and production use cases and not on providing an accessible educational environment for students learning GPU programming concepts for the first time.

Three.js [Thr], one of the most widely used JavaScript-based 3D graphics libraries, is actively developing WebGPU support, implemented through the `WebGPURenderer` that falls back to WebGL 2 when WebGPU is unavailable. In parallel, TSL (Three Shader Language), a JavaScript-based shader system, is being developed and compiled to WGSL and GLSL. While WebGPU support in Three.js is still under active development and therefore subject to change, its integration into such a significant library shows the growing importance of WebGPU as a future graphics API for the web.

Babylon.js [Bab] is a widely used, open-source, JavaScript-based 3D rendering engine that introduced WebGPU support with version 5.0 in May 2022 while maintaining full backward compatibility with the WebGL renderer, requiring only a change to the engine’s initialization. With version 8.0 in 2025, all core shaders of the engine were rewritten natively in WGSL, eliminating the need for runtime GLSL-to-WGSL conversion. This shows that WebGPU can be integrated into a mature production framework without disrupting existing workflows. However, due to its JavaScript focus and high level of abstraction, deeper GPU programming concepts remain hidden from developers.

PlayCanvas [Pla] is an open-source web game engine that first introduced WebGPU support in version 1.62 in March 2023. This required extensive internal restructuring to

2. Related Work

separate the rendering paths of WebGL and WebGPU and implement explicit rendering passes via a FrameGraph architecture. As of 2024, WebGPU support remains in beta and requires optional activation. However, most of the engine’s examples run on WebGPU with a fallback to WebGL 2. The engine highlights the architectural complexity of retroactively integrating WebGPU into an existing WebGL framework, as opposed to engines built from the ground up for WebGPU, and is primarily used for commercial purposes, leaving open the possibility of providing tools for GPU programming in the educational sector.

The Bevy Engine [Bev] is an open-source, data-driven game engine written in Rust which uses the wgpu library as its rendering backend and enables cross-platform GPU access on Windows, Linux, macOS, and the web via WebAssembly. Its entity-component system architecture offers an alternative to traditional scene graphs and the engine confirms the suitability of wgpu as a production rendering backend while offering insights into modern engine design patterns. However, due to its Rust-centric design and focus on game development rather than graphics education, C++-based WebGPU teaching tools with structured tutorials are still missing.

Gregory [Gre18] offers a comprehensive textbook on game engine architecture, covering rendering pipelines, resource management systems, physics integration, and scene graph design in detail. It is generally considered a standard work for understanding the structure of game engines, though it is often described as very complex, while online tutorials are considered more beginner-friendly. It does not cover WebGPU or modern explicit graphics APIs, and the representation of web-based deployment and browser environments is missing, meaning that the application of these patterns in the context of WebGPU training has not been studied yet.

Rohlf and Helman [RH94] presented IRIS Performer, a toolkit for visual simulation and real-time 3D graphics combining a low-level rendering library with a high-level hierarchical scene graph supporting pipelined parallel traversals across multiple CPUs, with graphics optimizations focused on efficient data transfer, state reduction, and mode setting minimization. The work addressed fundamental performance challenges of immediate-mode graphics and introduced the scene graph as a practical abstraction for managing rendering state and spatial hierarchies in real-time applications. While the architecture predates modern explicit GPU APIs, the scene graph concepts it established remain directly relevant to engine design and informed the hierarchical node system implemented in this thesis.

Unity Technologies [Uni19] introduced the Data-Oriented Technology Stack (DOTS), a set of packages including the Entity Component System, the C# Job System, and the Burst Compiler, designed to address performance limitations of Unity’s traditional object-oriented architecture by organizing data for cache-efficient processing. DOTS represents a significant shift in how game engines manage scene objects, prioritizing memory layout and CPU cache utilization over the developer familiarity of object-oriented scene graphs. While DOTS makes the performance advantages of ECS over traditional scene graphs visible, the added complexity makes it less suitable for introductory educational contexts, motivating the Unity-inspired scene graph approach taken in this thesis.

Caini [Cai18] developed EnTT, an open-source header-only C++ library implementing

2. Related Work

a fast and type-safe Entity Component System using sparse sets for cache-friendly component storage, widely adopted in game engines and simulations requiring high-performance entity management. The library shows that ECS architectures can be implemented cleanly in modern C++ and is a useful conceptual reference for understanding data-oriented alternatives to traditional scene graphs. However, the inherent complexity of ECS in terms of data layout and system scheduling makes it less approachable for students new to engine design, motivating the use of a more familiar scene graph architecture in this thesis.

Bilas [Bil02] presented an early and influential proposal for component-based game object systems at GDC 2002, arguing that traditional C++ inheritance hierarchies for game objects are fundamentally too rigid for the constantly changing requirements of game development, and proposing instead that objects be assembled from self-contained C++ and script components at runtime. Presented in the context of Dungeon Siege production, the talk argues that a data-driven component architecture enables designers to define object behavior with reduced engineering intervention for each content change. The proposal is an important precursor to later component-oriented and ECS-style engine designs and is relevant as context for the architectural decision in this thesis to use a scene graph over a component system, prioritizing familiarity for students over the flexibility of fully data-driven object composition.

Cook and Torrance [CT82] introduced the microfacet-based reflectance model that forms the theoretical foundation for modern physically-based rendering, treating surfaces as collections of microscopic facets with Beckmann distribution, Fresnel reflectance, and geometric attenuation to produce physically plausible specular highlights. Their work established the mathematical framework that underlies contemporary PBR implementations, though computational complexity delayed real-time adoption until GPU capabilities improved decades later. The engine implements a Cook-Torrance-based BRDF with GGX distribution, providing students with practical experience implementing the microfacet model in real-time shaders.

Schlick [Sch94] presented an efficient Fresnel reflectance approximation $F(\theta) = F_0 + (1 - F_0)(1 - \cos \theta)^5$ that replaced expensive calculations with a simple polynomial evaluation, which made real-time PBR practical on GPU hardware. The approximation became ubiquitous in game engines due to its minimal computational cost while mostly maintaining good visual accuracy across viewing angles. The engine implements Schlick's approximation in its PBR shaders, providing students with a working example of how theoretical optimization translates to practical shader code.

Walter et al. [WMLT07] introduced the GGX microfacet distribution function, producing specular highlights with longer tails that better match measured material reflectance than Beckmann or Blinn-Phong alternatives, alongside efficient importance sampling schemes making it practical for rendering applications. The GGX distribution has since become the standard in physically-based rendering pipelines and was notably adopted in Unreal Engine 4 [Kar13] as the foundation of its PBR shading model. The Vienna WebGPU Engine uses GGX as its normal distribution function, exposing students to the same physically-based approach used in modern production renderers.

Karis [Kar13] synthesized academic PBR theory into Unreal Engine 4's production shad-

2. Related Work

ing system by combining GGX distribution, Schlick’s Fresnel approximation, and Smith’s geometric shadowing, establishing the metallic-roughness workflow now standard across game engines. His metallic-roughness parameterization simplified artist workflows by reducing complex physical properties to two intuitive parameters: metallic controls dielectric versus conductor behavior, while roughness maps to microfacet distribution width. The engine adopts the same metallic-roughness parameterization, exposing students to the industry-standard PBR workflow used in production engines.

Burley [Bur12] introduced Disney’s Principled BRDF, providing intuitive artist parameters like subsurface, metallic, specular, roughness, and clearcoat instead of exposing complex physical properties such as index of refraction or microfacet distributions. The principled approach showed that physically plausible rendering and artist-friendly workflows are not mutually exclusive. While the engine implements a simplified metallic-roughness model rather than Disney’s full principled BRDF, the design philosophy of balancing physical correctness with artist accessibility directly influenced the material parameter design for educational clarity.

Pharr et al. [PJH16] provide the standard work on physically based rendering (PBR), covering the mathematical theory of light transport, reflection models, and material systems, and presenting a complete implementation of a professional renderer in literate programming. The book lays the theoretical foundations for PBR material systems, including the Cook-Torrance BRDF, microfacet distribution functions, and importance sampling, which underlie the material system implemented in this work. However, the focus remains on production rendering pipelines rather than real-time GPU rasterization, leaving the application of these principles to interactive WebGPU-based rendering as an area addressed by this thesis.

Akenine-Möller et al. [AMHH18] produced the 1,178-page Real-Time Rendering, Fourth Edition, the definitive reference for interactive graphics covering GPU architectures, rendering pipelines, physically-based shading, shadows, and modern techniques like clustered shading across DirectX 12, Vulkan, and OpenGL. While comprehensive for traditional graphics APIs, the 2018 edition predates WebGPU and provides no coverage of web-based GPU programming or browser deployment constraints. The engine implements many techniques described in Real-Time Rendering, including PBR workflows, shadow mapping with PCF, and multi-pass rendering, adapted specifically for WebGPU’s bind group architecture and targeted at educational contexts rather than production optimization.

Blinn [Bli78] introduced bump mapping for simulating surface detail through texture-based normal perturbation, enabling high-frequency surface appearance without additional geometry, and establishing the conceptual foundation for modern normal mapping techniques. The Vienna WebGPU Engine implements normal mapping in its PBR material system using tangent-space encoded normal maps, the modern GPU-based evolution of Blinn’s original approach. Bump mapping itself is considered outdated in contemporary real-time rendering, having been superseded by tangent-space normal mapping which operates directly in the fragment shader.

Williams [Wil78] introduced shadow mapping as a two-pass rendering technique, rendering a depth map from the light’s perspective and comparing fragment depths during

2. Related Work

shading to determine visibility. The Vienna WebGPU Engine implements shadow mapping for all supported light types: directional, spot, and point lights, building directly on this foundational algorithm. Aliasing artifacts and soft shadow filtering remained challenges addressed by later work such as Percentage Closer Filtering.

Reeves et al. [RSC87] introduced Percentage Closer Filtering to reduce shadow map aliasing by averaging multiple depth comparisons around each fragment rather than comparing a single texel, producing smoother shadow edges at minimal additional cost. PCF is fully implemented in the shadow mapping system of the Vienna WebGPU Engine across all light types. However, performance optimization of the shadow pipeline, including efficient sampling patterns, is left for future work.

Zhang et al. [ZSXL06] developed Cascaded Shadow Maps for large-scale environments by partitioning the view frustum into distance-based regions each with a dedicated shadow map, optimizing shadow map resolution allocation across varying distances from the camera. The Vienna WebGPU Engine fully implements cascaded shadow maps for directional lights, maintaining shadow quality across scenes of varying scale. Performance implications of cascaded shadow maps at higher cascade counts and automatic split parameter tuning remain open challenges.

Google’s Filament [Fil20] is an open-source C++ physically-based rendering engine targeting Android, iOS, Windows, Linux, macOS, and the web, exposing its C++ core to web developers through a WebAssembly module with a first-class JavaScript API compiled via Emscripten, and supporting multiple rendering backends including Vulkan, Metal, OpenGL ES, WebGL, and WebGPU. The engine provides a complete PBR implementation with an IBL pipeline, post-processing stack, and metallic-roughness material system, making it one of the few production C++ engines to support WebGPU as a native backend alongside more established APIs. While some tutorial material exists for the JavaScript web target, Filament lacks a structured learning progression for students approaching GPU programming from scratch in a native C++ context, so educational adoption remains outside its scope.

Unity’s Scriptable Render Pipeline [Uni22], introduced in 2018 with the High Definition Render Pipeline and Universal Render Pipeline as concrete implementations, exposes C# APIs for customizing Unity’s rendering at a lower level than the legacy built-in renderer, allowing developers to control render passes, culling, and draw calls through managed code. The system is an example of how abstraction layers can provide low-level control while maintaining cross-platform compatibility across DirectX, Vulkan, Metal, and OpenGL backends. However, SRP operates within Unity’s C#-based ecosystem with its own rendering abstractions, making it unsuitable as a teaching tool for understanding native WebGPU API design, shader programming in WGSL, or C++-based engine architecture that students encounter in graphics programming courses outside the Unity environment.

Godot Engine [God23] is a widely adopted open-source game engine that transitioned to modern explicit graphics APIs with version 4.0 in 2023, introducing Vulkan, Metal, and DirectX 12 backends alongside a compatibility renderer targeting WebGL 2, with experimental WebGPU support under active development. As an open-source engine with a

2. Related Work

large community and extensive documentation, Godot serves as a valuable reference for understanding how production engines abstract multiple graphics backends behind a unified rendering interface and structure high-level engine systems such as scene management, material workflows, and rendering passes. However, Godot’s documentation targets game developers using the engine rather than students learning underlying GPU programming concepts, and it offers neither low-level graphics education nor WebGPU-specific teaching material.

Dakkak et al. [DPH16] developed a web-based platform called WebGPU (unrelated to the modern W3C WebGPU specification) for teaching CUDA, OpenCL, and OpenACC programming through browser-based exercises with automatic grading and performance evaluation across GPUs from NVIDIA, AMD, and Intel. The platform showed that cloud-based GPU access enables scalable programming education without requiring students to maintain local GPU hardware, achieving deployment in massive open online courses with thousands of students. However, the system targeted GPGPU compute programming rather than graphics rendering, predates the modern WebGPU graphics API by several years, and does not address teaching structured graphics engine development or shader programming workflows relevant to contemporary web-based GPU graphics education.

Hmeljak and Zhang [HZ20] investigated replacing traditional computer graphics courses based on low-level APIs like OpenGL with project-based learning using commercial game engines, specifically Unity. Their results indicate that using a game engine as a software development framework lets graphics assignments cover core concepts such as shader-based programming without requiring students to manage low-level API boilerplate, keeping the focus on graphics fundamentals. This work is directly relevant as it validates the pedagogical approach of using an engine abstraction layer for teaching graphics programming, which the Vienna WebGPU Engine adopts by providing a structured C++ engine environment that exposes GPU concepts without requiring students to build the infrastructure themselves.

Humphrys [Hum24] developed Ancient Brain, a JavaScript-based educational platform specifically designed for teaching programming through graphics and interactivity. The platform includes teacher account management, code sharing, and thousands of student projects indicating high engagement. WebGPU integration is evolving, and systematic assessment of learning outcomes represents future research. Humphrys [Hum25] further explored integrating AI APIs into educational coding platforms, where students proved capable of building sophisticated applications combining graphics and AI. These efforts establish models for browser-based graphics education but leave WebGPU-specific curricula as future work.

De Vries [dV14] created Learn OpenGL, a comprehensive online tutorial series covering modern core-profile OpenGL from basic rendering to advanced techniques including deferred shading, shadow mapping, PBR, and post-processing, developed continuously since 2014 over 7 years and culminating in a physical print edition. The tutorial’s step-by-step structure with complete working code examples, clear explanations, and progressive difficulty has achieved widespread adoption as the de facto standard for learning OpenGL. While Learn OpenGL demonstrates an effective pedagogical model for teaching graphics

2. Related Work

APIs through practical implementation, its focus on OpenGL means WebGPU adaptation would require substantial rework to account for explicit resource management, bind group architecture, and WGSL shader language, representing a significant opportunity for WebGPU-focused educational content following a similar structured approach.

Sweller [Swe88] introduced the cognitive load theory and argued that conventional problem-solving can consume significant cognitive resources, leaving less memory capacity available for the acquisition of patterns, which in turn limits the learning effectiveness of beginners. This work combines computational modeling with experimental evidence supporting this mechanism. This provides the basis for guided instructional approaches in programming education, including step-by-step instructions that reduce unnecessary problem-solving load in early learning stages.

Berssanette and de Francisco [BdF22] conducted a systematic review of 33 studies examining Cognitive Load Theory in the teaching and learning of computer programming, which focuses on research published between 2010 and 2020. The review reports on the frequent use of worked-example strategies and subjective cognitive load measures across the surveyed literature. It further emphasizes that programming is a cognitively demanding field that requires the coordination of multiple skills, making instructional designs that reduce unnecessary cognitive load particularly relevant. These findings support the approach of a guided, step-by-step tutorial pursued in this study.

The Vienna Vulkan Engine [Hla24b] and Vienna Physics Engine [Hla24a], developed by Prof. Helmut Hlavacs at the University of Vienna, serve as educational platforms for teaching graphics programming and game development in courses such as Gaming Technologies and Cloud Gaming as part of the “Education, Didactics and Entertainment Computing” Research Group. These engines provide a proven model for educational rendering frameworks designed specifically for academic use, with multiple bachelor’s and master’s theses completed using it as a foundation. The engines prioritize educational clarity through consistent C++ patterns, CMake build systems familiar to students, and comprehensive documentation explaining both implementation and GPU concepts. While the Vienna Vulkan Engine targets the Vulkan API and has proven effective in university teaching contexts, the architectural patterns and pedagogical approaches pioneered by these engines directly informed the design philosophy of the Vienna WebGPU Engine, which continues the university’s approach to graphics education while adapting to emerging APIs.

3. Technical Foundations

This chapter outlines the foundational technologies, theories, and design patterns that the *Vienna WebGPU Engine* builds upon. The presented concepts represent established work by other researchers and practitioners that form the theoretical and practical basis for the contributions described in [Chapter 4](#). Understanding these foundations is essential for comprehending the architectural decisions and implementation strategies employed in the engine’s design.

The educational context for these foundations is shaped by the University of Vienna’s practice-oriented graphics curriculum, where students build and analyze engine subsystems in C++ through hands-on coursework and project-based formats [[Hla24b](#), [Hla24a](#)]. This context motivates a foundations chapter that emphasizes explicit API models, reusable architecture patterns, and conceptual transferability to modern rendering systems rather than one-off implementation details.

3.1. WebGPU Architecture, Programming Model, and Resource Management

The engine’s graphics architecture is built on WebGPU, a modern, cross-platform GPU API developed by the W3C GPU for the Web Community Group [[W3C23b](#)]. WebGPU represents a significant departure from legacy graphics APIs, embodying lessons learned from Vulkan [[Khr16](#)], Metal [[App14](#)], and DirectX 12 [[Mic15](#)]. Unlike its predecessor WebGL, which abstracted GPU details through an OpenGL-like implicit state machine, WebGPU exposes explicit resource management, command encoding, and pipeline configuration that mirror contemporary native APIs. This explicit design philosophy enables better performance optimization, clearer mental models of GPU operations, and more consistent cross-platform behavior. The specification defines a platform-agnostic surface interface that allows identical rendering code to target both web browsers and native desktop applications, addressing one of the major pain points in cross-platform graphics development where different platforms historically required separate rendering backends.

The implementation of the Vienna WebGPU Engine was primarily guided by Michel’s comprehensive tutorial series [[Mic22](#)], which provides practical patterns for WebGPU resource management, rendering pipelines, and C++ integration idioms. Michel’s work shows effective use of RAII principles for WebGPU resource lifetime management, leveraging C++ features such as namespaces, scoped enumerations, and default descriptor values to reduce boilerplate code while maintaining compatibility with the underlying C API. The tutorial introduces a lightweight C++ wrapper that transforms WebGPU’s C-style function calls into idiomatic C++ methods, significantly improving code readability and reducing common programming errors. For example, device buffer creation transforms from verbose C-style `wgpuDeviceCreateBuffer(device, &descriptor)` calls to concise C++

3. Technical Foundations

method invocations like `device.createBuffer(descriptor)`, while maintaining identical runtime performance. These patterns were adopted throughout the Vienna WebGPU Engine’s implementation to balance API accessibility with low-level control.

WebGPU’s command encoding model requires understanding how rendering operations are recorded into command buffers and submitted to GPU queues for execution. Commands are encoded using encoder objects that build command buffers, which are then submitted to the device queue for asynchronous GPU execution. This explicit model contrasts sharply with OpenGL’s implicit state machine, where draw calls immediately affect hidden driver state. The explicit approach provides several advantages: command buffers can be constructed on multiple CPU threads simultaneously, commands can be recorded once and replayed multiple times, and the separation between command recording and execution enables better driver optimization. However, this explicitness also increases the API surface area that developers must understand. The Vienna WebGPU Engine addresses this through its factory pattern architecture, which encapsulates common command encoding patterns while maintaining visibility into the underlying WebGPU operations for educational purposes.

Native WebGPU implementations exist in multiple forms, each with different maturity levels and platform support characteristics. Google’s Dawn [Goo26] provides a production-ready WebGPU implementation powering Google Chrome’s WebGPU support, offering excellent standards compliance and active development. The Rust-based `wgpu` library, available through `wgpu-native` bindings for C++, provides an alternative implementation with strong emphasis on safety and cross-platform consistency.

At the time of development, `wgpu-native` v0.19.4.1 was used as the native WebGPU implementation for the Vienna WebGPU Engine [gfx26]. This version was chosen for its comprehensive feature support, active community maintenance, and reliable behavior during the engine’s development period from early 2025 to February 2026. The project kept this baseline stable throughout development for reproducibility despite rapid ecosystem changes. Platform-specific differences in WebGPU implementations are isolated within initialization routines, allowing the majority of rendering code to remain platform-agnostic.

The Emscripten WebGPU ecosystem has undergone significant changes since the engine’s development period. The `emdawnwebgpu` port, maintained as a fork of Emscripten’s legacy bindings, was first introduced in Chrome 137 as Dawn’s official Emscripten binding [Chr25a], with streamlined integration added in Chrome 138 [Chr25b]. The standardized `webgpu.h` C API header was declared stable in Chrome 141, establishing a stable foundation for cross-platform WebGPU development [Chr25c]. Cross-platform deployment to web browsers through the Emscripten compilation toolchain is an important WebGPU capability, though full validation of web deployment with current tooling remains future work.

An important architectural difference between WebGPU and older APIs is its explicit validation and capability negotiation model. Applications first query adapter limits and optional features, then request a device configured for those capabilities. This sequence encourages developers to treat hardware variability as a first-class design constraint instead of assuming one fixed GPU profile. In educational contexts, this behavior is valuable

3. Technical Foundations

because it makes portability concerns visible at the beginning of implementation rather than only when deployment fails on another system. The validation layer also reports resource and pipeline misuse early, which improves debugging workflows for beginners who are still learning API contracts.

Pipeline state object design is another central concept inherited from modern explicit APIs such as Vulkan and DirectX 12 [Khr16, Mic15]. Rather than setting shader programs, blend modes, and depth states piecemeal at draw time, WebGPU packages most render state into immutable pipeline objects created up front. This approach front-loads configuration work but reduces runtime ambiguity and allows stronger driver optimization. It also helps students reason about rendering correctness, because pipeline definitions become declarative specifications of how draw calls should execute. The same conceptual model applies to compute pipelines, reinforcing the idea that graphics and compute workloads share one coherent resource and execution framework.

Asynchronous execution semantics further distinguish WebGPU from immediate-mode APIs. Command recording occurs on the CPU, while actual execution happens later on GPU queues, and many operations such as buffer mapping complete through asynchronous callbacks or promise-based mechanisms. This encourages non-blocking application structure and introduces students to latency-hiding techniques that are essential in modern graphics and GPGPU workflows. At the same time, it requires explicit thinking about lifetime and ownership: resources used by in-flight command buffers must remain valid until execution completes. Understanding this execution timeline is a prerequisite for diagnosing stutter, synchronization bugs, and data hazards in larger rendering systems.

Building on this architectural foundation, the engine relies on WebGPU’s explicit resource management and binding model to organize GPU data and shader inputs. WebGPU’s resource management model differs fundamentally from legacy OpenGL’s global state machine, requiring explicit declaration of all GPU resources and their relationships through bind groups. Understanding this binding model proves essential for effective WebGPU programming, as incorrect resource organization leads to verbose shader code, redundant GPU updates, and difficult-to-maintain pipelines. The Vienna WebGPU Engine’s architecture embraces WebGPU’s explicit binding philosophy while providing factory abstractions that reduce common boilerplate without obscuring the underlying resource model. This section establishes foundational concepts for GPU buffer management, bind group organization, and shader-driven resource declaration that inform the engine’s implementation described in [Chapter 4](#).

Buffer objects represent the primary mechanism for CPU-to-GPU data transfer in modern graphics APIs. WebGPU defines several buffer usage types, each optimized for specific access patterns and update frequencies. Uniform buffers store small, frequently-updated constant data accessible to shaders during pipeline execution, typically containing per-frame camera parameters, per-object transformation matrices, or material properties. WebGPU imposes alignment requirements on uniform buffer bindings, commonly requiring 256-byte alignment for dynamic offsets, necessitating careful structure padding in shader code. The engine addresses these alignment constraints through explicit structure layout in WGSL shader declarations, ensuring CPU-side data structures match GPU memory

3. Technical Foundations

layout requirements. Uniform buffers support rapid updates through queue-based write operations that avoid explicit staging buffers, enabling efficient per-frame data streaming for dynamic scene parameters.

Storage buffers provide larger, more flexible GPU memory regions supporting both read and write access from compute and fragment shaders. Unlike uniform buffers with strict size limitations, storage buffers can contain arbitrarily large data arrays, making them suitable for light arrays with runtime-determined counts, particle systems with variable population sizes, or procedural geometry generation. WebGPU’s support for runtime-sized arrays in storage buffers eliminates the need for shader variants based on array sizes, allowing a single shader to handle varying light counts without recompilation. The syntax `struct LightArray { lights: array<Light> }` in WGSL declares such runtime-sized arrays, with actual element counts determined at bind group creation time. This flexibility simplifies shader management compared to APIs requiring compile-time array sizes or uniform buffer packing tricks to simulate dynamic arrays.

Vertex buffers supply per-vertex attribute data to the graphics pipeline’s vertex processing stages. WebGPU supports flexible vertex buffer layouts including interleaved attributes where position, normal, tangent, UV, and color data appear consecutively for each vertex, and non-interleaved layouts separating attribute types into distinct buffers. Interleaved layouts generally provide better cache locality during vertex fetch operations, as accessing one vertex’s attributes retrieves adjacent data from the same cache line. The engine’s vertex structure definition in C++ mirrors WGSL shader input declarations through `@location` attributes, ensuring consistent vertex attribute interpretation between CPU mesh data and GPU vertex shaders. The `WGPUVertexBufferLayout` descriptor explicitly defines stride, step mode, and per-attribute formats, disambiguating vertex data interpretation without implicit driver assumptions.

Index buffers store triangle connectivity information, referencing vertex buffer elements to define mesh topology. Using indexed rendering reduces vertex duplication in triangle meshes, particularly for closed surfaces where each vertex participates in multiple triangles. WebGPU supports 16-bit and 32-bit index formats, with format selection trading off maximum vertex count against memory usage. The engine employs 32-bit indices through `WGPUIndexFormat_Uint32` to support large meshes without vertex count limitations, accepting modest memory overhead for implementation simplicity. Index buffer creation follows the same factory pattern as other buffer types, with explicit usage flags distinguishing index data from uniform or storage buffers.

Bind groups organize related GPU resources into logical units that shaders can reference collectively. The WebGPU binding model uses a two-level hierarchy: bind group layouts define abstract resource slot declarations specifying buffer types, texture dimensions, and sampler configurations, while bind groups provide concrete resource instances bound to those slots. Pipelines reference bind group layouts during creation, establishing which resource configurations the pipeline expects. This separation between layout and instance enables resource updates without pipeline recompilation, as changing which specific texture binds to a material slot requires only creating a new bind group with identical layout, not rebuilding the entire graphics pipeline. The pattern mirrors Vulkan’s descriptor set

3. Technical Foundations

model while providing simpler syntax and reduced ceremony compared to Vulkan’s verbose descriptor pool allocation.

Shader-driven bind group generation represents an important architectural pattern that the Vienna WebGPU Engine employs extensively. Rather than manually declaring bind group layouts in C++ code separate from shader implementations, the engine parses WGSL source code for `@group(N)` and `@binding(M)` annotations during shader compilation. The shader factory extracts these declarations to automatically generate corresponding `WGPUBindGroupLayout` objects, ensuring bind group layouts precisely match shader expectations without manual synchronization effort. This approach eliminates entire categories of runtime errors where shader code expects resources at different binding slots than the application provides. When shaders change, layouts regenerate automatically, maintaining consistency between shader resource declarations and engine-side bind group creation.

The engine organizes bind groups by update frequency following established best practices from Vulkan and DirectX 12 development. Group 0 contains per-frame data updated once per rendered frame, typically camera matrices, viewport dimensions, and global time values. Group 1 holds per-view lighting data including light arrays and shadow map textures, updated when lighting configuration changes. Group 2 stores per-object transformation matrices updated for each draw call. Group 3 contains material properties and textures, updated when materials change but cached across objects sharing materials. This frequency-based organization minimizes redundant GPU state changes, as bind groups with stable contents remain bound across multiple draw calls while only frequently-changing groups require rebinding. The pattern shows how architectural decisions at the API level influence performance characteristics and teaches students that resource organization strongly affects rendering efficiency beyond raw algorithmic complexity.

Texture binding within bind groups requires consideration of texture types, sampling modes, and synchronization requirements. WebGPU distinguishes sampled textures used for reading during shading, storage textures supporting read-write compute shader access, and render target textures receiving rasterization output. Shadow mapping exemplifies texture usage transitions: shadow maps serve as depth render targets during shadow pass execution, then transition to sampled textures during main rendering when fragment shaders sample them for visibility tests. WebGPU’s implicit resource state tracking eliminates manual barrier insertion for these transitions, simplifying rendering code compared to Vulkan’s explicit image layout management. However, developers must still understand conceptual texture usage patterns to diagnose issues where texture data appears unavailable or corrupted due to incorrect usage flag specification.

Comparison samplers represent specialized sampler objects that WebGPU provides specifically for shadow mapping and depth testing operations. When bound to depth textures, comparison samplers automatically perform hardware-accelerated depth comparison during texture sampling, returning binary visibility results rather than raw depth values. This hardware support enables Percentage Closer Filtering through multiple comparison samples averaged by shader code, significantly improving shadow quality without manual depth comparison implementations. The sampler comparison function specified during

3. Technical Foundations

sampler creation determines whether depth tests use less-than, greater-than, or equal comparison semantics, configuring hardware behavior for different depth buffer conventions. The engine uses these specialized samplers throughout its shadow mapping implementation, exposing students to modern GPU features that simplify algorithms which previously required complex shader code.

Buffer update strategies balance simplicity against performance optimization. The most straightforward approach uses `queue.writeBuffer()` to directly transfer data from CPU memory to GPU buffers, appropriate for uniform buffers updated once per frame with modest data sizes. Larger updates or high-frequency modifications benefit from staging buffers that batch multiple writes before GPU submission, reducing synchronization overhead between CPU and GPU. For buffers accessed by both compute and graphics pipelines, careful attention to usage flags prevents validation errors where buffers declared without appropriate access permissions cause pipeline binding failures. The engine’s buffer factory encapsulates these usage flag combinations, providing specialized creation methods for uniform, storage, vertex, and index buffers that automatically configure appropriate usage flags for each buffer type’s intended access patterns.

Understanding these resource management and binding concepts establishes the foundation for implementing the Vienna WebGPU Engine’s shader-driven architecture described in [Chapter 4](#). The explicit binding model initially appears verbose compared to OpenGL’s implicit state management, but this explicitness enables better performance optimization, clearer mental models of GPU resource relationships, and more maintainable rendering code as project complexity grows. Students learning WebGPU through the engine encounter these concepts gradually through factory abstractions before progressing to direct bind group manipulation. In this way, the teaching framework scaffolds complex API understanding through architectural layering.

Limit management is a practical aspect of the binding model that often determines portability. WebGPU devices expose constraints such as maximum bind groups, per-stage resource counts, uniform buffer binding sizes, and texture dimension bounds. Designing shader interfaces without considering these limits can lead to architectures that work on one desktop GPU but fail on integrated or mobile-class hardware. Therefore, robust systems typically budget binding slots early and group resources by lifetime and update frequency to remain within common hardware envelopes. This planning process teaches an important engineering lesson: API correctness alone is insufficient when cross-platform deployment is a project requirement.

Another foundational concept is layout compatibility and pipeline reuse. Because pipelines are created against explicit bind group layouts, frequent layout churn causes redundant pipeline creation and increased startup latency. Many engines therefore define stable layout conventions for frame, view, object, and material data that remain consistent across multiple shaders. This enables shader specialization without changing higher-level binding contracts and reduces runtime state fragmentation. The principle mirrors interface versioning in software architecture and is directly relevant to educational engines, where predictable conventions reduce cognitive load for newcomers.

WGSL memory layout rules also deserve attention when discussing resource binding.

3. Technical Foundations

Host-shareable types and alignment constraints require CPU-side structures to match shader-side padding and field ordering exactly. Otherwise, buffers appear corrupted despite successful API calls. These issues are especially common when students first combine vectors, matrices, and arrays in uniform blocks. Documenting layout contracts and validating offsets during development are therefore standard practices in professional graphics code. The broader lesson is that explicit APIs shift many hidden driver decisions into application code, rewarding disciplined data modeling and verification.

3.2. Physically-Based Rendering Theory

Physically-based rendering has become the dominant shading paradigm in real-time graphics, replacing ad-hoc shading models with approaches grounded in the physics of light transport. The Vienna WebGPU Engine implements PBR using the Cook-Torrance microfacet BRDF model [CT82], which describes surface reflectance as the statistical behavior of microscopic surface facets. This model provides a principled framework for representing how light interacts with surfaces based on surface roughness and material properties. The Cook-Torrance BRDF consists of diffuse and specular components, where the specular term models mirror-like reflections from microfacets using a normal distribution function, a geometric attenuation term, and a Fresnel term describing reflectance at different viewing angles.

The specific formulation implemented in the engine follows Karis’s influential work on real-time PBR for Unreal Engine 4 [Kar13], which built upon Disney’s principled BRDF research [Bur12]. Karis demonstrated that the GGX normal distribution function [WMLT07], also known as Trowbridge-Reitz distribution, produces more realistic specular highlights than earlier alternatives like Blinn-Phong, particularly for rough surfaces. The GGX distribution exhibits a longer tail than other normal distributions, better matching observed reflectance behavior of real materials. For Fresnel approximation, the implementation uses Schlick’s approximation [Sch94], which provides visually acceptable results at significantly lower computational cost than full Fresnel equations. These choices balance physical accuracy with real-time performance constraints, following industry best practices established by Epic Games and adopted widely across game engines.

The metallic-roughness workflow employed by the engine simplifies material authoring by reducing material properties to artist-friendly parameters. Metallic values interpolate between dielectric and metallic Fresnel responses, while roughness controls the spread of specular highlights through the GGX distribution’s alpha parameter. This parameterization, popularized by Disney [Bur12] and standardized through glTF material specifications, enables artists to create physically plausible materials without deep understanding of optical physics. For comprehensive mathematical derivations of BRDF theory and Monte Carlo light transport, readers are referred to the authoritative text by Pharr et al. [PJH16], which provides rigorous treatment of physically-based rendering algorithms.

Beyond selecting a BRDF model, physically based shading requires respecting energy conservation and reciprocity constraints. In practical terms, a material should not reflect more radiant energy than it receives, and view/light exchange should behave symmetric-

3. Technical Foundations

ally under the same geometric conditions. Real-time PBR workflows approximate these properties using normalized distribution functions, visibility terms that limit multi-bounce amplification, and Fresnel terms constrained to plausible ranges. Even when approximations are used for performance, these constraints are critical because they make material behavior stable under changing illumination. This stability is one of the main reasons physically based pipelines have replaced ad-hoc shading models in production tools and teaching material.

Image-based lighting complements direct-light BRDF evaluation by integrating illumination from environment maps, allowing plausible reflections and ambient response without requiring dense light rig setups. Modern real-time pipelines commonly use prefiltered environment maps and BRDF lookup tables to approximate the specular integral efficiently, an approach popularized in game-engine practice and documented by Karis [Kar13]. Diffuse irradiance is typically represented through low-frequency convolution of the environment, while specular response uses roughness-aware prefiltering. These approximations are foundational for understanding why contemporary PBR scenes appear coherent across different lighting conditions. They also show how physically motivated models are translated into tractable GPU-friendly algorithms.

High dynamic range rendering and tone mapping are closely coupled with PBR workflows. Physically plausible lighting produces luminance ranges far above display-referred output, so a mapping from scene-referred radiance to monitor-referred color space is required for presentation. Common operators such as Reinhard or filmic mappings compress highlights while preserving mid-tone contrast, and gamma-correct output conversion ensures predictable display behavior. Although tone mapping is often presented as a post-processing step, conceptually it belongs to the same foundation as BRDF modeling because both determine how physical light quantities become visible appearance. For students, this linkage clarifies why “correct” shading still looks wrong when output transfer and exposure control are ignored.

Normal mapping [Bli78] extends surface detail beyond geometric mesh resolution by perturbing surface normals during shading computation. The technique stores per-pixel normal deviations in texture maps, allowing high-frequency surface detail without corresponding geometric complexity. The Vienna WebGPU Engine implements tangent-space normal mapping [PAC97], which encodes normals relative to the surface’s local tangent frame rather than world space. Tangent-space encoding provides several advantages: normal maps remain valid under object transformations, texture coordinates directly drive normal lookups, and compression can exploit the constrained range of tangent-space vectors. The implementation constructs TBN matrices from mesh vertex tangents and bitangents, transforming tangent-space normals from texture data into world space for lighting calculations. This approach follows standard game engine practice as documented in real-time rendering literature [AMHH18].

Shadow mapping [Wil78] determines visibility between surface points and light sources by rendering depth maps from light perspectives and comparing fragment depths during final rendering. The technique’s fundamental limitation involves aliasing artifacts where shadow map resolution proves insufficient for scene detail. The engine addresses these

3. Technical Foundations

artifacts through Percentage Closer Filtering [RSC87], which samples multiple shadow map texels around each fragment and averages binary visibility tests, producing soft shadow boundaries instead of harsh aliasing. For directional lights covering large scene extents, Cascaded Shadow Maps [ZSXL06] partition the view frustum into distance-based regions, each with dedicated shadow maps at appropriate resolutions. Nearby geometry receives high-resolution shadows while distant regions use lower resolution, optimizing shadow map memory allocation. Point light shadows employ cube map textures with six shadow map faces covering all directions, enabling omnidirectional shadow casting at the cost of increased memory usage and rendering overhead.

3.3. Scene Graph and Entity Organization

Scene graphs provide hierarchical organization of virtual environments, enabling intuitive spatial relationships through parent-child node structures. The Vienna WebGPU Engine’s scene graph architecture draws inspiration from Unity’s GameObject-Component system and Godot’s Node hierarchy, balancing educational familiarity with architectural flexibility. Gregory’s comprehensive treatment of game engine architecture [Gre18] provides foundational patterns for scene graph implementation, including transform hierarchy management, update order traversal, and render submission strategies. Early scene graph systems such as SGI’s IRIS Performer [RH94] established core concepts of hierarchical spatial organization, though modern implementations have evolved significantly to address cache coherency and parallel processing requirements.

Transform hierarchies enable child nodes to inherit parent transformations, creating intuitive spatial relationships where moving a parent automatically moves all children. The engine implements Unity-style transforms with lazy matrix updates, recalculating world-space transformation matrices only when local transforms or parent relationships change. This lazy evaluation pattern, documented extensively in game engine architecture literature [Gre18], reduces redundant matrix multiplications compared to naive per-frame recalculation. Transform components store position, rotation, and scale in local space, computing world-space matrices through recursive parent multiplication when requested. Rotation representation uses quaternions rather than Euler angles to avoid gimbal lock and enable smooth interpolation, following best practices for 3D transformations. The decomposition of transformation state into separate position, rotation, and scale components simplifies animation and physics integration compared to storing raw transformation matrices.

Node lifecycle methods provide hooks for gameplay logic and rendering preparation. The engine implements `start()`, `update()`, and `lateUpdate()` callbacks mirroring Unity’s event functions, giving scene objects predictable update ordering. The `start()` method initializes node state after construction and scene attachment, `update()` handles per-frame logic and input response, and `lateUpdate()` performs post-update processing such as camera following. This event-driven lifecycle pattern, common across modern game engines, provides clear mental models for when different types of logic should execute. Render traversal occurs separately from update traversal, collecting visible objects for

3. Technical Foundations

GPU submission without interleaving gameplay state changes with rendering operations.

Alternative entity organization approaches exist, particularly Entity Component Systems that optimize cache locality and enable data-oriented design. Unity’s DOTS framework [Uni19] and Michele Caini’s EnTT library [Cai18] demonstrate ECS approaches that store components in contiguous memory arrays rather than hierarchical node structures. These systems excel for entity-heavy games with thousands of active objects, where cache-friendly iteration significantly improves performance. However, ECS architectures introduce conceptual complexity that can hinder educational goals. Students familiar with object-oriented programming find scene graphs more intuitive, as parent-child relationships and component composition map naturally to familiar programming patterns. The Vienna WebGPU Engine prioritizes this educational clarity over maximal performance optimization, accepting scene graph overhead as acceptable for typical rendering workloads.

Spatial acceleration strategies are a second foundational aspect of scene organization. Even in scene-graph-centric engines, rendering performance depends on efficiently rejecting objects that do not contribute to the current frame. Frustum culling removes objects outside camera visibility, while additional schemes such as spatial partitioning grids, bounding volume hierarchies, or portal systems reduce traversal cost in complex scenes [AMHH18]. These methods are independent of a specific engine implementation and appear across graphics literature because they address a universal scaling problem: scene complexity typically grows faster than per-frame CPU/GPU budgets. Understanding culling as a data-management problem, not only a rendering problem, provides important conceptual context for later architecture choices.

Update scheduling also influences scene-system behavior and determinism. Game engine literature distinguishes variable-step update loops, fixed-step simulation loops, and hybrid approaches where simulation and rendering run at different rates [Gre18]. Variable-step updates are simple but can introduce frame-rate-dependent behavior, while fixed-step simulation improves stability for animation and physics at the cost of interpolation complexity. Educational engines often start with a simplified variable-step model, then gradually introduce timestep management as students encounter consistency problems. Presenting these trade-offs in the foundations chapter helps readers interpret why scene systems are structured around explicit lifecycle phases and frame contexts.

3.4. Design Patterns for Graphics Systems

The Vienna WebGPU Engine employs several established design patterns from software engineering and game engine development to manage complexity and improve code maintainability. The Factory Pattern from Gamma et al.’s seminal work [GHJV94] provides the architectural foundation for GPU resource creation. Abstract factories encapsulate creation logic behind consistent interfaces, allowing client code to request resources without understanding platform-specific implementation details. The engine’s `WebGPUContext` serves as an abstract factory providing access to concrete factories for different resource types: `TextureFactory`, `MaterialFactory`, `MeshFactory`, `ShaderFactory`, and `PipelineFactory`. Each factory encapsulates the verbose WebGPU descriptor initializ-

3. Technical Foundations

ation and resource lifetime management required for GPU object creation, presenting simplified interfaces to application code.

This factory architecture mirrors approaches used in production game engines, particularly Epic’s Rendering Hardware Interface [Epi23] that abstracts differences between DirectX, Vulkan, and Metal implementations. By isolating platform-specific rendering code within factory implementations, the pattern enables engine-level rendering algorithms to remain agnostic of underlying graphics APIs. While the Vienna WebGPU Engine currently targets only WebGPU, the factory pattern’s indirection provides extensibility for future backends or WebGPU implementation variants. The educational benefit lies in showing how large-scale graphics systems manage complexity through architectural patterns rather than monolithic implementations.

Handle-based resource management decouples CPU-side resource metadata from GPU-side resource lifetime, addressing common difficulties in graphics programming where dangling pointers or use-after-free errors can cause crashes or corruption. The engine implements a typed handle system where `Handle<Texture>`, `Handle<Mesh>`, and similar types provide type-safe references to resources managed by corresponding resource managers. Handles store unique identifiers and version numbers, enabling detection of stale references when resources are reloaded or destroyed. This approach follows patterns popularized by Bilas’s data-driven game object systems [Bil02] and refined in contemporary engines like Our Machinery [Gra18]. Gregory’s game engine architecture text [Gre18] provides comprehensive coverage of handle systems and their advantages over raw pointer-based resource management.

The handle system enables hot-reloading of assets during development, a critical feature for iterative graphics programming. When a shader file changes, the engine can reload the shader, recreate affected pipelines, and update all materials referencing those pipelines without requiring application restart. Handles remain valid across reloads because they reference logical resources rather than GPU memory addresses. This pattern illustrates how indirection, often viewed as performance overhead, provides valuable flexibility for development workflows. The educational value lies in showing students how production engines balance runtime efficiency with development productivity through architectural decisions.

Resource lifetime management in WebGPU requires careful attention to object dependencies and destruction ordering. WebGPU objects maintain internal reference counts, and destroying parent objects before children can cause validation errors or crashes. The factory pattern encapsulates these lifetime concerns, ensuring resources are destroyed in correct order when factories are shut down. Additionally, factories can implement caching strategies to avoid redundant GPU resource creation, though the current implementation favors simplicity over aggressive caching. Understanding these lifetime patterns proves essential for students working with explicit graphics APIs where automatic garbage collection does not manage GPU resources.

Cross-platform abstraction layers represent another critical design pattern for graphics engines. The engine isolates platform-specific code within initialization routines while maintaining platform-agnostic rendering paths. The `WebGPUContext::initialize()`

3. Technical Foundations

method handles differences between native window surface creation via SDL and web canvas binding through Emscripten, but subsequent rendering code operates identically across platforms. This separation of concerns follows principles documented in cross-platform engine architectures like bgfx [Kar23], which abstracts multiple graphics APIs behind uniform interfaces. The pattern teaches students how architectural discipline enables code reuse across deployment targets, avoiding the maintenance burden of parallel platform-specific implementations.

Additional software patterns from established engineering practice further improve maintainability in graphics codebases. The Observer Pattern [GHJV94] is frequently used to propagate state changes such as window resize events, material updates, or hot-reload notifications without tight coupling between subsystems. Command Pattern variants are commonly applied in editor tooling, undo/redo stacks, and render-graph scheduling, where actions must be recorded, replayed, and inspected. Data-driven configuration through external asset metadata reduces recompilation cycles and supports reproducible content pipelines. These patterns are not specific to one engine, but they are repeatedly adopted in production and research systems because they separate concerns and localize complexity.

Testing and diagnostics strategies are equally important at architecture level. Unit testing of pure CPU-side systems (math, serialization, resource indexing) can be automated in conventional CI pipelines, while GPU behavior is often validated through golden-image comparisons, shader validation layers, and deterministic replay captures. Logging frameworks and runtime debug UIs provide observability for state-heavy rendering systems where failures may only appear under specific frame timings or resource combinations. In educational projects, these diagnostics have additional value because they externalize hidden engine state and make abstract graphics concepts inspectable. Consequently, design patterns should be evaluated not only by runtime performance but also by how well they support debugging, teaching, and iterative development.

The combined application of these patterns, with factories for resource creation, handles for resource management, and platform abstraction for deployment flexibility, creates a coherent architecture that manages complexity while maintaining educational transparency. Each pattern addresses specific challenges in graphics programming: factories simplify verbose API usage, handles enable asset reloading and prevent common errors, and abstraction layers support cross-platform deployment. Students studying the engine can observe how these patterns compose into larger architectural structures, providing models for their own graphics projects. The patterns represent established solutions from decades of game engine development, distilled into an educational implementation that prioritizes clarity over maximal performance optimization.

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

This chapter presents the Vienna WebGPU Engine, an educational rendering engine designed to teach modern GPU programming concepts through WebGPU. The engine addresses the research questions posed in [Chapter 1](#) by assessing WebGPU’s current state as a cross-platform graphics API compared to established low-level alternatives (RQ1), proving the feasibility of building a fully functional rendering engine with modern rendering features (RQ2), and presenting educational design patterns that effectively teach GPU programming concepts to learners (RQ3). Development began in early 2025 and continued until February 2026 using wgpu-native v0.19.4.1 as the WebGPU implementation, which was intentionally kept stable throughout the thesis work to avoid disruptions from API changes. The implementation focuses on a layered architecture that combines multiple design patterns, namely factories for resource creation, handles for type-safe references, managers for lifecycle coordination, and caching for performance, while maintaining the power and flexibility of modern graphics APIs. Development revealed important findings about WebGPU’s implementation landscape during this period, including cross-platform behavior differences, tooling maturity gaps, and web deployment obstacles that inform the evaluation of its readiness for educational use.

The engine’s implementation began with Michel’s “LearnWebGPU: WebGPU C++ Guide” tutorial series [[Mic22](#)] as the foundational codebase, which provided the initial project structure and WebGPU initialization patterns. The tutorial’s approach to WebGPU context setup, device configuration, and C++ wrapper design formed the starting point for development, offering learners a well-documented entry into WebGPU programming. However, the engine diverges significantly from the tutorial’s architecture by introducing factory patterns, resource managers, caching systems, and a complete scene graph. These extensions are specifically designed to address educational clarity at scale. The tutorial introduces individual WebGPU concepts through isolated examples, while the engine integrates these concepts into a cohesive architecture capable of supporting complex multi-object scenes with materials, lighting, and shadows. This architectural divergence reflects the thesis goal of teaching not just WebGPU usage but modern rendering engine design patterns suitable for educational contexts.

The decision to implement the engine in C++ rather than using Rust-based APIs directly reflects educational priorities within the University of Vienna’s Entertainment Computing curriculum. The familiarity with C++ development enabled faster implementation and more confident architectural decisions during the thesis timeframe. Within the University of Vienna, C++ remains the primary language for graphics and game engine courses taught by Prof. Helmut Hlavacs, including Gaming Technologies and Cloud Gaming, which utilize the Vienna Vulkan Engine [[Hla24b](#)] for educational purposes. Multiple bachelor’s and master’s theses have been completed using this established C++ engine,

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

which speaks for its pedagogical effectiveness. The Vienna WebGPU Engine continues this tradition, providing learners with skills directly applicable to existing university courses while introducing modern WebGPU concepts. Using Rust would have required learners to learn both a new graphics API and a new systems programming language simultaneously, increasing cognitive load beyond acceptable levels for educational contexts. The C++ approach enables learners already familiar with the language from prerequisite courses to focus exclusively on GPU programming concepts without language barrier distractions.

The C++ wrapper from Michel’s tutorial proved valuable for improving code readability throughout the engine implementation. The wrapper provides RAII-style resource management and type-safe enumerations that transform verbose C API calls into more natural C++ design without runtime overhead. However, the engine’s implementation revealed situations where the wrapper’s const-correctness enforcement conflicts with WebGPU’s C API requirements, particularly during buffer mapping operations and descriptor construction where the C API expects non-const pointers. In these specific cases, the engine drops down to direct C API calls rather than fighting the wrapper’s type system. This pragmatic approach teaches learners that abstractions are tools for clarity rather than strict rules. When an abstraction introduces more complexity than it removes, working around it can be good engineering judgment rather than failure. The wrapper remains the default interface for most engine code, providing educational clarity and safety, while direct C API calls are used only in specific cases where the wrapper’s design does not align with WebGPU’s requirements.

4.1. WebGPU Platform State and Implementation Landscape

The choice of WebGPU implementation for the engine required careful consideration of educational priorities, build system complexity, and API stability. [Table 4.1](#) compares the major WebGPU implementations available during the engine’s development period. The decision to use `wgpu-native v0.19.4.1` rather than Google’s Dawn was motivated by integration simplicity. At the time of development in early 2025, `wgpu-native` provided stable pre-compiled binaries integrating via a standard `CMakeLists.txt`, whereas Dawn offered no official pre-built binaries and required compiling thousands of source files, resulting in significantly longer initial build times and a more complex setup process that would present an unnecessary barrier for learners trying to get started with the engine. Dawn does provide superior validation error messages and is generally ahead of `wgpu-native` in specification compliance, and has since introduced official prebuilt binaries as of September 2025, leaving migration to Dawn as a viable future improvement. Dawn’s build system is based on GN (Generate Ninja), a Google-internal meta-build system with little adoption outside the Google ecosystem, which would require learners to learn an additional unfamiliar toolchain. In contrast, `wgpu-native` integrates directly with CMake, allowing learners to use build tools they already know from other C++ courses. Notably, `wgpu-native` itself has also progressed to `v27.0.4.0` as of December 2025, leaving migration to a more current release as additional future work. This build system accessibility aligns with the overall educational goal of minimizing setup overhead so learners can focus on

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

the concepts of GPU programming rather than toolchain configuration.

Table 4.1.: Comparison of WebGPU implementations considered for the engine

Implementation	Language	Platform Support	Build System	Maturity
wgpu-native v0.19.4.1	Rust (C bindings)	Windows, Linux	macOS, CMake-friendly	Educational/ Beta
Google Dawn	C++	Windows, Linux, Chrome	macOS, GN (depot_tools)	Production- ready
Browser Native	JavaScript/ TypeScript	Web (Chrome, Firefox, Edge)	N/A (browser)	Production- ready
Emscripten (-sUSE_WEBGPU)	C/C++ to WASM	Web compilation	Emscripten	Deprecated
Emscripten (emdawnwebgpu)	C/C++ to WASM	Web (Chrome 137+)	Emscripten 4.0.10+	Active

The decision to freeze the engine on wgpu-native v0.19.4.1 throughout the entire development period was a practical constraint during engine development. WebGPU implementations evolved rapidly during 2025, with API changes and behavioral adjustments appearing in new releases that would have required significant refactoring of engine internals. Continuously tracking these changes would have derailed development progress, forcing constant rewrites of working systems to accommodate API modifications rather than advancing the engine’s feature set and educational content. By pinning to a specific version, development effort could focus on implementing rendering features, creating tutorials, and refining architectural patterns without the distraction of chasing moving API targets. The stability provided by version pinning also carries educational benefits as tutorials and documentation remain accurate without requiring continuous updates to match evolving API syntax, reducing confusion for learners working through examples months after their creation. Educational software development often requires trading access to the latest features for the stability needed to actually complete the project.

Web deployment through Emscripten represented one of the most significant challenges encountered during development. Initial implementation successfully compiled and ran in web browsers using Emscripten’s `-sUSE_WEBGPU` flag, which provided WebGPU bindings for native C++ code compiled to WebAssembly. This early success suggested straightforward cross-platform deployment would be achievable. However, Emscripten’s maintainers subsequently deprecated the `-sUSE_WEBGPU` implementation in favor of the `emdawnwebgpu` port, which integrates Dawn-based WebGPU bindings directly into Emscripten 4.0.10 and later versions. The `webgpu.h` C API header reached stability in Chrome 141, released at the end of Q3 2025 [Chr25c], providing a standardized interface for future development. The engine’s web deployment work remains incomplete due to these ecosystem changes occurring mid-development, a concrete example of the practical challenges of building on rapidly evolving standards. For the thesis scope, Windows-native deployment using wgpu-native proved sufficient to demonstrate all GPU programming concepts without the additional complexity of maintaining web-compatible builds. Future work should address full migration to the `emdawnwebgpu` path once the toolchain stabilizes further.

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

Cross-platform testing revealed implementation differences across GPU backends and windowing systems that inform educational discussions about API portability. The most significant GPU-level issue involved bind group layout validation strictness on macOS Metal backends. The engine’s initial implementation worked correctly on Windows using the Direct3D 12 backend but failed validation on macOS with Metal due to stricter ordering requirements: Metal requires all bind group layout slots up to the highest used index to be explicitly provided, filling unused intermediate indices with `nullptr` to preserve correct group index alignment. Omitting intermediate slots causes validation errors where shader resource bindings cannot be found in the pipeline layout. Windows seems to allow bind group layouts where intermediate slots are omitted, hiding this portability issue during initial development. The engine was updated to generate bind group layouts in strict index order, ensuring cross-platform compatibility.

Platform-specific windowing behavior presented another portability challenge. On Linux, SDL2’s Wayland backend exhibited different initialization requirements compared to X11, necessitating explicit backend selection logic with fallback mechanisms when preferred backends were unavailable. This showed that cross-platform graphics development extends beyond GPU API differences to encompass windowing systems, input handling, and operating system integration layers. Each of these layers requires platform-specific testing to identify subtle behavioral variations.

These experiences illustrate the challenges of writing truly portable graphics code. True cross-platform compatibility requires testing on all target platforms and adhering to the strictest interpretation of API specifications rather than relying on permissive behavior from a single implementation. Issues that appear as bugs on one platform may represent correct enforcement of specification details that other platforms ignore, making comprehensive multi-platform testing essential during development rather than as a final validation step.

Device capability querying emerged as another critical cross-platform consideration during implementation. Different GPU vendors and driver versions support varying maximum texture sizes, bind group counts, and buffer sizes that must be queried at runtime. The engine’s initialization sequence allows for custom device limits to be specified, but by default it queries the actual device limits using `wgpuAdapterGetLimits()`. If these limits are not compatible with the specified custom limits it will clamp those and a warning is logged for each limit that is clamped. This approach ensures that the engine can run on a wide range of hardware configurations without crashing due to unsupported resource sizes, while also making hardware capabilities visible to learners during startup. This could be further extended in future work so that engine features are also automatically enabled or disabled based on hardware capabilities. For example, the shadow mapping system queries the maximum texture dimension limits to determine appropriate shadow map resolutions rather than hardcoding 2048x2048 sizes that might exceed capabilities on integrated GPUs. This flexible approach gives learners a practical example of how to query and adapt to device capabilities in modern graphics programming.

4.2. Layered Architecture, Handle System, and Scene Graph

As the Vienna WebGPU Engine was built based on the foundational codebase provided by Michel’s “LearnWebGPU: WebGPU C++ Guide” tutorial series [Mic22], the initial architecture was a simple project made up of the following files: `main.cpp`, `Application.cpp`, and `ResourceManager.cpp` as well as their corresponding headers. The architecture evolved significantly from this starting point, growing into a layered design that organizes code into distinct subsystems responsible for different aspects of rendering and resource management. The architecture can be roughly divided into the following layers:

The **Core Layer** provides foundational infrastructure used throughout the engine. At its heart lies the `Identifiable<T>` template base class, which assigns unique 64-bit IDs to all resources, enabling the type-safe `Handle<T>` system for resource references. The inheriting `Versioned` class adds counters to track resource modifications, allowing GPU resources to detect when their CPU counterparts have changed and require updates. The `PathProvider` utility centralizes file path resolution for assets, shaders, and resources across different build configurations. This layer establishes patterns that prevent common errors such as dangling pointers and use-after-free bugs by enforcing handle-based access with validation through `std::optional<T>` returns. While the handle system serves as the primary resource reference mechanism throughout the engine, certain subsystems employ `std::shared_ptr` and `std::weak_ptr` for lifetime management where observer patterns are needed. The `PipelineManager` specifically uses weak pointers to pipeline objects, enabling hot-reload detection by checking whether the weak pointer has expired. When a pipeline is no longer referenced elsewhere in the engine, the weak pointer becomes invalid, signaling that dependent resources can be safely recompiled and updated without breaking active rendering operations. This mixed approach is pragmatic: handles provide efficient type-safe references for most resources, while weak pointers offer lifetime observation for hot-reloading pipelines without introducing circular dependencies. Future refactoring could unify these mechanisms by extending the handle system with observer capabilities, eliminating smart pointers entirely while maintaining hot-reload functionality through handle-based lifetime queries. More details on the handle system are provided later in this section.

The **Resource Management Layer** sits on top of the core layer and manages all CPU-side asset data through specialized manager classes. The `ResourceManagerBase<T>` template provides common functionality for registration, lookup, and lifecycle management that specialized managers inherit. The `TextureManager` loads image files via `TextureLoader` using the `stb_image` library and maintains `Texture` objects containing pixel data in CPU memory. The `MeshManager` stores vertex positions, normals, texture coordinates, tangents, and index data in `Mesh` objects, while the `MaterialManager` maintains `Material` definitions with PBR properties and texture slot references. The `ModelManager` coordinates these components, loading complete 3D models through `ObjLoader` for Wavefront OBJ files or `GltfLoader` for glTF and GLB formats, automatically handling static mesh and material assignments. Animation import is not yet supported. The aggregate

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

ResourceManager class provides a unified interface to all sub-managers, exposing methods like `loadModel()`, `loadTexture()`, and `createMaterial()` that learners use without managing individual manager instances. Again, handles from the primary resource reference path across these managers, as detailed below, which enables safe resource access patterns and features like hot-reloading where CPU resources persist while GPU resources are recreated.

The **Rendering Layer** bridges CPU resources and GPU execution through the **WebGPUContext** class, which serves as the central interface providing access to the WebGPU device, queue, surface, and all factory instances. The factory pattern is employed extensively through the **BaseWebGPUFactory<SourceT, ProductT>** template, from which specialized factories derive. The **WebGPUTextureFactory** converts **Texture** handles into **WebGPUTexture** objects containing the `wgpu::Texture`, `wgpu::TextureView`, and `wgpu::Sampler` instances, automatically determining optimal texture formats, creating appropriate views, and uploading pixel data to the GPU via `queue.writeTexture()`. The **WebGPUMeshFactory** generates vertex and index buffers from **Mesh** data, while the **WebGPUMaterialFactory** creates complete material bind groups with property uniform buffers and texture bindings. The **ShaderFactory** employs a builder-based approach for bind group layout specification, where developers explicitly declare shader resource requirements through method calls such as `addBindGroup()`, `addStorageBuffer()`, and `addFrameBindGroup()` that mirror the shader's actual resource usage, as detailed in [Section 4.3](#). The **ShaderRegistry** maintains all shader instances with hot-reloading support, allowing runtime shader modification without application restart, as implemented in the `reloadAllShaders()` method that iterates through registered shaders and triggers recompilation. The **PipelineManager** builds on this foundation by caching render pipelines with their associated **PipelineConfig** structures containing shader references, vertex layouts, depth-stencil state, blend modes, and primitive topology, enabling pipeline recreation when shaders change while preserving configuration.

The **Scene Graph Layer** implements a hierarchical node system inspired by Unity's **GameObject** architecture, making the engine immediately familiar to learners with prior game development experience. The **Node** base class establishes parent-child relationships through `addChild()` and `removeChild()` methods, providing tree traversal via `visitPreOrder()` and `visitPostOrder()` for systematic scene processing. Each node contains a **Transform** component storing local position, rotation (as quaternion), and scale, with lazy evaluation of world matrices through dirty flags to avoid unnecessary matrix multiplications. When `setLocalPosition()` is called, the transform marks itself dirty, and subsequent `getWorldMatrix()` calls recompute the transformation hierarchy only when needed, propagating from parent to child. Specialized node types extend this base: **UpdateNode** adds the `update(deltaTime)` method for game logic and animations, **RenderNode** provides `render(RenderContext)` for drawing operations, and **ModelRenderNode** combines both interfaces to render 3D models with automatic material binding and mesh submission. The **LightNode** represents light sources (directional, spot, point) with shadow mapping configuration, while **CameraNode** encapsulates view and projection matrices with support for perspective and orthographic modes. Lifecycle

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

methods mirror Unity’s MonoBehaviour pattern: `start()` initializes nodes when added to the scene, `update()` and `lateUpdate()` handle per-frame logic, `preRender()` prepares GPU resources, `render()` submits draw commands, `postRender()` performs cleanup, and `onDestroy()` releases resources when nodes are removed. The `Scene` class orchestrates this lifecycle, maintaining a root node that always exists and a default camera node created automatically, managing the frame loop by visiting all nodes in the appropriate order for each phase.

An alternative to this design would have been an Entity-Component-System architecture, which is popular in modern engines for flexibility and cache-friendly processing. However, for this thesis, the scene graph was chosen as it reduces conceptual overhead for learners who are already learning WebGPU concepts, bind group organization, and rendering pipeline mechanics at the same time. The hierarchical model also maps naturally to common educational examples such as parented camera rigs, light attachments, and grouped scene objects, where transform inheritance is a core teaching point. This decision prioritizes architectural readability and pedagogical clarity over raw data-oriented optimization, which aligns with the educational scope of the engine.

The scene update order is intentionally explicit: update and late-update logic run before rendering, and renderable nodes are then collected for pass execution in the renderer. This sequencing provides deterministic behavior for learners, because gameplay-side state changes are finalized before draw submission begins. In practice, collected render items will be organized by material and pipeline to reduce state changes across draw calls, which makes batching behavior visible without requiring learners to write low-level scheduling code themselves. The result is a teaching-friendly frame model in which node logic, transform propagation, and render submission remain clearly separated but easy to trace from scene graph code to GPU commands.

The **Application Layer** ties all subsystems together through the `GameEngine` class, which initializes the window via SDL, creates the WebGPU device and surface through `WebGPUContext`, sets up the `InputManager` for keyboard and mouse events, initializes the `Renderer` with the multi-pass pipeline, and starts the `ImGuiManager` for debug UI overlays. The main game loop processes SDL events, updates input state, advances physics simulation, invokes scene lifecycle methods (`update()`, `lateUpdate()`), triggers rendering passes, presents the frame to the swap chain, and maintains frame timing for delta time calculations. The `EngineContext` pattern provides nodes with read-only access to engine subsystems without circular dependencies: `engine()->input()` returns the `InputManager`, `engine()->gpu()` provides the `WebGPUContext`, `engine()->resources()` accesses the `ResourceManager`, and `engine()->scenes()` retrieves the `SceneManager`. This context propagates automatically when nodes are added to the scene graph, allowing custom node implementations to query input state, load assets, access GPU factories, or switch scenes without storing subsystem pointers directly. The `SceneManager` handles scene creation, loading, and switching, calling `onDestroy()` on all nodes in the previous scene before invoking `start()` on nodes in the newly loaded scene, enabling structured state transitions between game menus, gameplay, and other modes.

The **Handle System** functions as the cross-layer reference mechanism that allows

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

resources to be safely accessed across the core and resource layers. Assets originate in resource managers, are consumed by scene nodes during update and render preparation, and are finally translated into GPU representations by factories in the rendering layer. A pure ownership-based pointer model would couple lifetime decisions to usage sites, making this cross-layer path difficult to reason about and fragile during reloads. The engine therefore uses a dedicated handle mechanism as the canonical non-owning link across layers, while managers retain ownership and validate lookups. The following block details this mechanism.

Therefore, the `Handle<T>` template class (Listing 4.1) is implemented through type-safe 64-bit IDs rather than raw pointers, preventing dangling pointer bugs and enabling hot-reloading without invalidating existing references. Each handle contains only an ID and no direct pointer, making it lightweight (8 bytes), trivially copyable, and serializable for save systems. The resolution mechanism works through a static resolver function installed by each `ResourceManagerBase<T>` specialization: when `handle.get()` is called, it invokes the resolver which looks up the resource in the manager’s internal registry and returns `std::optional<std::shared_ptr<T>`. This indirect access pattern forces validation at every use site. Learners cannot assume that a handle remains valid across frames, which teaches defensive programming practices essential for game development where resources may unload dynamically.

The handle system coexists with direct `std::shared_ptr<T>` and `std::weak_ptr<T>` usage throughout the engine, creating a hybrid ownership model that serves different purposes. The `ResourceManagerBase<T>` template provides the foundation for all specialized managers (`TextureManager`, `MeshManager`, `MaterialManager`, `ModelManager`), storing resources in `std::unordered_map<Handle<T>, std::shared_ptr<T>` where handles serve as keys for internal organization. The manager supports multiple lookup strategies: `get(handle)` for handle-based access returning `std::optional<std::shared_ptr<T>`, `getByID(id)` for direct ID lookup, `getByName(name)` for named resource retrieval, and `getAllWithName(name)` for resources sharing a name. Removal is equally flexible through `remove(handle)` or `remove(shared_ptr)`, with the manager invalidating the handle upon removal. While managers internally use handles to organize resources, the external interface allows retrieval of shared pointers that learners can store directly for continued use without needing to call `handle.get()` repeatedly, avoiding resolver overhead on every access while guaranteeing the resource persists. The `TextureManager` illustrates this dual storage: `std::unordered_map<std::string, TextureHandle> m_imageCache` for path-to-handle lookup, and the base class’s `std::unordered_map<Handle<Texture>, std::shared_ptr<Texture>` `m_resources` for handle-to-resource lookup. This allows `getTextureByPath(path)` to first retrieve the handle from the image cache, then use that handle to get the shared pointer from the base manager, or directly return the cached shared pointer. The handle thus serves as an indirection key rather than the sole reference mechanism. This separation prevents circular dependencies common in scene graphs: a parent node can hold a `Handle<Material>` without worrying that destroying the handle will prematurely release the material if other nodes also reference it.

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

```
1 template <typename T>
2 class Handle {
3 public:
4     using id_type = uint64_t;
5
6     constexpr Handle() = default;
7     explicit constexpr Handle(id_type id) : m_id(id) {}
8
9     [[nodiscard]] constexpr id_type id() const { return m_id; }
10    [[nodiscard]] constexpr bool valid() const { return m_id != 0; }
11
12    // Resolver pattern: indirect lookup via static function
13    [[nodiscard]] std::optional<std::shared_ptr<T>> get() const {
14        if (!valid() || !s_resolver)
15            return std::nullopt;
16        return s_resolver(*this);
17    }
18
19 private:
20     friend class ResourceManagerBase<T>;
21
22     // Manager installs resolver for this resource type
23     static void setResolver(
24         std::function<std::optional<std::shared_ptr<T>>(Handle<T>>>
25         resolver
26     ) {
27         s_resolver = std::move(resolver);
28     }
29
30     id_type m_id = 0; // 0 = invalid/null handle
31     static inline std::function<
32         std::optional<std::shared_ptr<T>>(Handle<T>)
33     > s_resolver;
```

Listing 4.1: Handle System with Resolver Pattern

Version tracking for CPU-to-GPU synchronization relies on the **Versioned** mixin present in most CPU resource classes (**Texture**, **Material**, **Mesh**). Each resource carries a generation counter accessed via `getVersion()` that increments when the resource is modified. Calling `texture->setPixelData()` or `material->setAlbedo()` bumps the version number, signaling that any cached GPU representations are potentially stale. The responsibility for detecting version mismatches and triggering GPU resource updates falls on the consumers of GPU resources, not the factories themselves. Render nodes, rendering passes, or higher-level managers must store both the `Handle<T>` and the last-known version number they used to create the GPU resource, compare against `resource->getVersion()` each frame or before rendering, and call the factory again when a mismatch is detected. This pattern enables lazy hot-reloading: resources only regenerate GPU data when actually needed for rendering and only when their CPU counterparts have changed, avoiding unnecessary work for resources not currently visible or in use.

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

The current implementation exhibits mixed lazy-reloading behavior depending on resource type and usage context. Some GPU resources perform full recreation when version changes are detected. `WebGPUTextureFactory::createFromHandle()` uploads complete pixel data and recreates the entire `wgpu::Texture`, `wgpu::TextureView`, and associated descriptors even if only texture filtering or wrapping mode changed. Other systems implement partial updates: material property buffers created during shader initialization can be updated in-place via `queue.writeBuffer()` when `MaterialProperties` values change, without recreating the entire bind group if texture references remain unchanged. The `PostProcessingPass` applies render-time lazy evaluation by storing `std::weak_ptr<WebGPUPipeline> m_pipeline` and checking if `(auto pipeline = m_pipeline.lock())` during `render()`. If the weak pointer has expired (pipeline destroyed during shader hot-reload), the pass calls `getOrCreatePipeline()` to regenerate it, otherwise it reuses the cached pipeline. This mixed approach reflects the current state of the engine where full lazy reloading with granular change detection is identified as future optimization work: GPU resources should detect exactly what changed in their CPU counterparts (pixel data, sampler settings, bindgroup values, texture settings) and try to update only the affected GPU state (buffer writes, descriptor updates) rather than performing complete resource recreation. As of now this is not fully implemented as it partially updates more than necessary in some cases and performs full recreation in others, but the version tracking system is in place to enable such optimizations in the future without changing the public interface of factories or resource managers.

GPU resource factories implement handle-based caching to avoid redundant GPU resource creation. The `WebGPUTextureFactory` maintains `std::unordered_map<Handle<Texture>, std::shared_ptr<WebGPUTexture>> m_textureCache` where the handle serves as the lookup key. When `createFromHandle(handle)` is called, the factory first checks the cache. If a GPU texture exists for that handle, it returns the cached instance immediately without recreating GPU resources. This caching strategy means that multiple calls with the same handle (from different render nodes, materials, or passes) reuse the same GPU texture, buffer objects, and views, avoiding memory waste and upload bandwidth. However, this cache does not track CPU resource modifications automatically. The factory simply stores handle-to-GPU-resource mappings without monitoring version changes. The `ResourceManagerBase<T>` template provides the foundation for all specialized managers, implementing registration, lookup, and the resolver installation that connects handles to resources. During manager construction, it calls `Handle<T>::setResolver()` with a lambda capturing `this`: `[this](Handle<T> h) { return this->lookup(h.id()); }`, establishing the link between handle IDs and the manager's internal storage. The lookup method returns `std::optional<std::shared_ptr<T>>`, forcing callers to handle missing resources explicitly, which has educational benefits: learners immediately see when resources fail to load and the optional return prevents null pointer dereferences that plague C++ beginners.

The aggregate `ResourceManager` class provides a unified interface to all sub-managers through member variables `m_textureManager`, `m_meshManager`, `m_materialManager`, and `m_modelManager`, along with loaders `m_objLoader`, `m_gltfLoader`, and `m_textureLoader`.

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

High-level methods like `loadModel(path)` coordinate multiple managers: the `ObjLoader` parses the file, extracts mesh data (vertices, normals, UVs, indices), discovers material references in the accompanying MTL file, calls `m_textureManager->loadTexture()` for each diffuse and normal map, creates `Material` objects with property structs and texture handles via `m_materialManager->createPBRMaterial()`, registers meshes with material assignments in `m_meshManager`, and finally constructs a `Model` object containing mesh collections that `m_modelManager` registers and returns as `Handle<Model>`. This orchestration remains hidden from learners, who simply call one method and receive a handle representing the complete loaded model, ready to pass to `WebGPUModelFactory::createFromHandle()` for GPU upload.

Future extensions to the versioning system should introduce fine-grained change flags alongside generation counters. The `Versioned` mixin could maintain a `uint32_t m_changeMask` where individual bits represent specific property categories (`PIXELS_CHANGED = 1 << 0`, `SAMPLER_CHANGED = 1 << 1`, `FORMAT_CHANGED = 1 << 2`), allowing GPU factories to query `texture->getChangeFlags()` and perform targeted updates: if only `PIXELS_CHANGED` is set, call `queue.writeTexture()` to upload new data without recreating the texture object. If `SAMPLER_CHANGED` is set, regenerate only the sampler and associated bind groups while preserving texture allocations. Material updates could differentiate between `PROPERTIES_CHANGED` (requiring buffer write but not bind group recreation) and `TEXTURES_CHANGED` (requiring new bind groups but not buffer recreation), significantly reducing GPU resource churn during iterative content authoring. The factory cache maps would extend from `std::unordered_map<Handle<T>, std::shared_ptr<WebGPUResource>` to include version tracking: `std::unordered_map<Handle<T>, std::pair<std::shared_ptr<WebGPUResource>, uint64_t>` where the second element stores the last-observed version, enabling factories to automatically detect stale cache entries and perform incremental updates without requiring consumers to manually track versions. This design would maintain the educational simplicity of “call the factory when you need a GPU resource” while achieving production-quality performance through transparent optimization of unchanged resources.

The current handle implementation has additional limitations that future work should address. The resolver pattern uses global static state per type (`Handle<Texture>::s_resolver`), preventing multiple independent managers for the same resource type and complicating unit testing where isolated manager instances are desirable. Handles provide no access tracking, so there is no way to query “which handles reference this resource?” or “when was this handle last accessed?”, information crucial for streaming systems that unload unused assets to save memory. A robust production handle system would extend `Handle<T>` with observer capabilities for lifetime monitoring, access pattern tracking (read count, last access timestamp), and invalidation callbacks (notify when resource unloads), while keeping the educational simplicity of the current tutorials.

The educational rationale for handles over raw pointers or pure shared pointers becomes clear when examining learner code patterns in the tutorials together with the architectural ordering described above. Tutorial 01 (Unlit Shader) has learners load a texture via `auto handle = textureManager->loadTexture("diffuse.png")` and pass it directly to the

material system without managing pointer ownership or lifetime details. The engine resolves the handle to a valid resource or returns `std::nullopt` for graceful failure handling. Tutorial 02 (Custom Bind Groups) then demonstrates explicit validation in `if (auto tex = m_tileTextureHandle.get()) { /* upload to GPU */ }`, reinforcing defensive access patterns without introducing pointer complexity. This design prevents common failure modes such as unchecked load failures, stale references after resource removal, and circular ownership between scene nodes and resources. At the same time, advanced usage remains possible through manager methods such as `getResource(id)` that return `std::optional<std::shared_ptr<T>`. The key educational decision is therefore to sequence complexity rather than remove it: subsystem responsibilities are made explicit first, the handle bridge separates ownership from access second, and factory abstractions encapsulate repetitive descriptor ceremony third. This ordering keeps the CPU-to-GPU flow traceable while moving repeated setup logic out of call sites and into reusable manager, handle, and factory components. In this sense, abstraction serves as staged guidance, not concealment, and responsibility-level evidence for this shift is documented in [Appendix A.2](#). Building on this foundation, the next section examines how shader and pipeline automation applies the same principle to bind group and pipeline creation.

4.3. Shader Reflection, Pipeline Automation, and Multi-Pass PBR Rendering

Shader development with WGSL provided a clear foundation for implementing the engine's PBR rendering pipeline. The language's explicit resource binding syntax using `@group` and `@binding` attributes makes resource organization immediately visible in shader code, reinforcing the mental model of bind groups as shader-visible resource containers. Within the engine, shaders are in the `resources` directory and the most necessary shaders are implemented as part of the engine itself, while additional shaders are provided as part of the tutorial content. The standard rendering shader is the `PBR_Lit_Shader.wgsl` which implements physically-based rendering with support for albedo, roughness, metallic properties, normal mapping, and shadow receiving. Additionally shaders for engine internal use were created, such as `shadow2D.wgsl` for shadow map generation or `mipmap_blit.wgsl` for GPU-based mipmap generation, and `fullscreen_quad.wgsl` for rendering multiple render targets onto the screen. Shaders organize resources using WGSL's binding attributes, with `@group(0)` conventionally reserved for per-frame data such as camera matrices and time values. This convention reflects the performance cost of bind group swaps, where keeping frequently accessed frame data at a consistent location minimizes GPU pipeline stalls. However, bind group ordering remains flexible and ultimately depends on the shader author's organizational preferences and specific rendering requirements.

The bind groups represent GPU resources required by the shader, and the pipeline layout defines how these resources are organized and accessed during rendering. Manually managing these GPU resources presents a substantial amount of boilerplate and synchronization challenges which obstruct learning. Without automation, developers must manually create bind group layouts by specifying every re-

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

source's binding index, visibility flags (vertex or fragment shader), resource type (uniform buffer, texture, sampler), and buffer sizes. Each binding requires a minimum of 5 lines of descriptor configuration. For example, a single uniform buffer binding requires setting `entry.binding`, `entry.visibility`, `entry.buffer.type`, `entry.buffer.minBindingSize`, and `entry.buffer.hasDynamicOffset`, while texture bindings additionally require `texture.sampleType`, `texture.viewDimension`, and `texture.multisampled` configuration. Developers must then assemble these individual entries into bind group layout descriptors, create the pipeline layout from multiple bind group layouts, configure the render pipeline with vertex input state, depth-stencil settings, blend modes, and primitive topology, and finally create bind groups that match the layouts with actual GPU resources.

To illustrate the verbosity, consider the shadow mapping shader used for 2D directional and spot light shadows (`shadow2d.wgsl`), which requires two bind groups: group 0 contains shadow-specific uniforms (light view-projection matrix), and group 1 contains per-object uniforms (model matrix). Creating just the bind group layout for group 0 requires extensive descriptor configuration (Listing 4.2), and this is before even creating the actual bind group instance, allocating the uniform buffer, or beginning pipeline layout construction.

```
1 // Manual WebGPU code to create bind group layout for shadow uniforms
2 // This creates ONLY the layout for group 0 (shadow uniforms)
3
4 // Step 1: Define the uniform buffer binding entry (5 required fields)
5 wgpu::BindGroupLayoutEntry shadowUniformEntry{};
6 shadowUniformEntry.binding = 0; // Binding index (@binding(0))
7 shadowUniformEntry.visibility = WGPUShaderStage_Vertex;
8 shadowUniformEntry.buffer.type = wgpu::BufferBindingType::Uniform;
9 shadowUniformEntry.buffer.hasDynamicOffset = false;
10 shadowUniformEntry.buffer.minBindingSize = sizeof(ShadowPass2DUniforms);
11
12 // Step 2: Create bind group layout descriptor (aggregates entries)
13 wgpu::BindGroupLayoutDescriptor layoutDesc{};
14 layoutDesc.entryCount = 1; // Number of bindings in this group
15 layoutDesc.entries = &shadowUniformEntry; // Pointer to entry array
16
17 // Step 3: Actually create the bind group layout object
18 wgpu::BindGroupLayout shadowBindGroupLayout
19     = device.createBindGroupLayout(layoutDesc);
```

Listing 4.2: Manual Bind Group Layout Creation for One Uniform Buffer

The manual approach exposes the core problem: each binding requires five descriptor fields minimum, the bind group layout creation requires assembling descriptors into arrays, and this sequence creates only the layout for group 0. It does not allocate the actual uniform buffer, does not create a bind group instance that links the buffer to the layout, and does not address additional bind groups which require repeating the entire process. Creating the complete render pipeline requires constructing the pipeline layout from both bind group layouts, compiling the shader module from the WGSL file, configuring the render pipeline descriptor with vertex input attributes (position only for shadow pass),

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

fragment output formats (depth-only), primitive topology (triangle list), depth-stencil state (depth write enabled, depth compare function Less), multisample state, and blend state, resulting in substantial boilerplate before any actual rendering can occur.

WGSL lacks runtime reflection capabilities that would enable automatic resource discovery from compiled shaders. The `device.createShaderModule()` function compiles WGSL source code but provides no API to query resource bindings, buffer sizes, or pipeline requirements from the compiled module. The shader module becomes opaque after compilation. Unlike SPIR-V which embeds reflection metadata accessible through libraries like SPIRV-Cross, WGSL modules offer no introspection mechanism. At the time of development in early 2025, no publicly available C++ libraries existed for WGSL shader reflection that could parse source code and extract binding information automatically. Implementing a complete WGSL parser would require building a lexer, parser, semantic analyzer, and binding extractor. This would be a substantial standalone project consuming months of development time that falls outside the thesis scope, which focuses on rendering engine architecture and educational design rather than language tooling development. Without reflection tooling available, manual specification of shader resources becomes necessary but must be simplified to remain educationally tractable for learners without extensive GPU programming experience.

```
1  auto shaderInfo = m_context.shaderFactory()  
2      .begin(  
3          shader::defaults::SHADOW_PASS_2D,  
4          ShaderType::Unlit,  
5          PathProvider::getResource("shadow2d.wgsl"),  
6          "vs_shadow", "fs_shadow",  
7          VertexLayout::Position  
8      )  
9      .addBindGroup(  
10         bindgroup::defaults::SHADOW_PASS_2D,  
11         BindGroupReuse::PerFrame,  
12         BindGroupType::ShadowPass2D  
13     )  
14     .addCustomUniform("uShadow", sizeof(ShadowPass2DUniforms),  
15         WGPUShaderStage_Vertex)  
16     .addObjectBindGroup() // Engine-provided standard object uniforms  
17     .build();
```

Listing 4.3: ShaderFactory Builder Pattern for Shadow Pass Shader

The engine addresses this through a builder pattern implemented in `ShaderFactory`, where developers declaratively describe shader resource requirements using fluent API calls. Rather than writing verbose bind group layout descriptors directly, learners call standardized methods like `addFrameBindGroup()` for per-frame resources, `addObjectBindGroup()` for per-object uniforms, `addMaterialBindGroup()` for material properties with textures, and `addShadowBindGroup()` for shadow pass data, or define custom bind groups via `addBindGroup("CustomName")` followed by resource specifications. The engine uses semantic names rather than numeric indices to identify bind groups: `"FrameBindGroup"` for camera and time uniforms, `"ObjectBindGroup"` for model matrices,

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

"MaterialBindGroup" for PBR properties and textures, and "LightBindGroup" for lighting data and shadow maps. When the `WebGPUMaterialFactory` needs to create material bind groups, it calls `shader->getBindGroupLayout("MaterialBindGroup")` to retrieve the layout, ensuring it binds resources to the correct group regardless of the numeric index assigned during pipeline creation. This naming approach works better than index-based access because the shader can choose any order for the groups, and the engine determines the correct layout based on the semantic name, rather than relying on hard-coded group numbers that may change as the shader is further developed. Ordering and especially grouping still matter for performance (keeping frequently accessed resources in lower group numbers and grouping frequently accessed resources together), but the API abstracts away the need for learners to manage this directly, allowing them to focus on declaring resource needs rather than optimizing bind group indices.

The builder pattern (Listing 4.3) provides standardization and metadata annotations that enable engine-level automation. The factory translates these high-level declarations into appropriate `wgpu::BindGroupLayoutEntry` descriptors automatically, determining visibility flags, binding indices within groups, and resource types from the method calls. The critical advantage emerges from standardized bind group methods: calling `addObjectBindGroup()` injects the engine's standard object uniform layout (model and normal matrices) automatically, ensuring that shadow pass shaders use the exact same object data layout as PBR shaders, debug shaders, and all other geometry-rendering shaders. This standardization means the renderer can create one object bind group per renderable and use it across all render passes without per-shader bind group variations. Learners do not manually configure object uniforms for each shader because the engine provides them consistently. The metadata annotations (`BindGroupReuse`, `BindGroupType`) provide critical hints enabling the renderer to automatically determine "this shader needs frame data, object data, and shadow-specific uniforms" and create appropriate bind groups.

The builder pattern still requires manual synchronization between WGSL shader code and C++ declarations, as changing `@binding(0)` to `@binding(1)` in the shader file requires updating the builder code to match. Nevertheless, it significantly reduces the surface area for errors by centralizing resource declarations in a high-level API rather than scattering verbose descriptor configuration throughout the codebase. The educational benefit comes from reduced cognitive load: learners focus on declaring what resources their shader needs (uniform buffers, textures, samplers) rather than wrestling with WebGPU descriptor structs, binding indices, visibility flags, and buffer size calculations that obscure the actual GPU programming concepts being taught.

Pipeline creation proceeds automatically from the bind group layouts declared through the builder API. The `ShaderFactory` collects all bind group layouts specified via `addBindGroup()` calls and assembles them into a `wgpu::PipelineLayoutDescriptor`, setting bind group layouts in order (group 0, group 1, etc.) and providing this layout to the `PipelineManager` when creating render pipelines. When `PipelineManager::getOrCreatePipeline(config)` is called, it checks the pipeline cache using the configuration hash as the key. If a cached pipeline is found, it is returned immediately. Otherwise, the manager constructs a new `wgpu::RenderPipelineDescriptor` by

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

combining the shader’s pipeline layout with the vertex layout, depth-stencil state, blend settings, and primitive configuration from the provided `PipelineConfig`, then creates and caches the pipeline for reuse. This automation means learners specify pipelines through high-level configuration structures rather than constructing WebGPU descriptor chains manually, while the caching system ensures efficient pipeline reuse across multiple draw calls and frames.

Future work should focus on implementing true shader reflection to eliminate manual builder synchronization entirely. A dedicated WGSL parser could extract resource bindings directly from shader source code, generating bind group layouts automatically without requiring C++ declarations to mirror shader structure. As the WebGPU ecosystem matures, publicly available reflection libraries may emerge providing functionality similar to SPIRV-Cross for Vulkan, enabling automatic introspection of compiled shader modules. Such reflection would ensure perfect synchronization by construction: adding a uniform buffer or texture to a shader would automatically create the corresponding bind group layout entry without touching engine code, eliminating synchronization errors entirely. Beyond reflection, the builder pattern itself could benefit from improved consistency. Currently, some bind groups use semantic names while others use type enums, and metadata annotations lack systematic validation. The engine already provides a suitable foundation for these improvements: `ShaderRegistry` supports hot-reloading and caching, and the named bind group approach enables refactoring without breaking existing code. Until reflection tooling becomes available and stable in the C++ WebGPU ecosystem, the current builder pattern represents a pragmatic compromise between educational clarity and implementation feasibility, providing substantial automation benefits over raw WebGPU API usage.

Beyond bind groups and pipelines, the same abstraction strategy extends to textures, buffers, and texture views, which also require substantial boilerplate in raw WebGPU. Creating a texture from pixel data typically involves descriptor setup, GPU texture creation, texture-view creation, and sampler creation. The engine encapsulates this sequence through factory methods such as `WebGPUTextureFactory::createFromHandle(handle)`, which perform descriptor configuration, resource creation, and cache lookup behind a single call. The method checks for an existing cached resource for the handle, creates it only when missing, and returns the cached instance otherwise, which reduces repeated setup work across render passes. The helper-aware counting boundary for these operation-level examples is documented in [Appendix A.2](#).

The engine defines a hierarchy of factories for different GPU resource types, all inheriting from a common `BaseWebGPUFactory<SourceT, ProductT>` template that provides shared caching and retrieval functionality. Specialized factories include `WebGPUTextureFactory`, `WebGPUPipelineFactory`, `WebGPUBindGroupLayoutFactory`, and others for buffers and texture views. The latter two are also used internally for bind group and pipeline creation as part of the shader automation path described earlier in this section. Each factory implements resource-specific creation logic while reusing the base class caching strategy to avoid redundant GPU allocations. For example, `WebGPUTextureFactory` maintains a cache mapping `Handle<Texture>` to `std::shared_ptr<WebGPUTexture>`, ensuring that

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

multiple requests for the same texture handle return the same GPU resource instance, optimizing memory usage and upload bandwidth. The factories also implement lazy creation and hot-reloading support: if a GPU resource is requested but the underlying CPU resource has changed (detected through version tracking), the factory can automatically recreate the GPU resource on demand, ensuring that the renderer always uses up-to-date data without manual synchronization at call sites.

The factory-based approach also provides a clear resource flow from disk to GPU. When a model is loaded from disk, the `ObjLoader` or `GltfLoader` parses the file and creates CPU-side resource objects (e.g., `Texture`, `Mesh`, `Material`) which are registered with the appropriate resource managers and assigned handles. These are used inside the `ModelManager` to create a `Model` object containing mesh collections with material assignments. When the renderer needs to draw a model, it calls the corresponding factory methods (e.g., `WebGPUModelFactory::createFromHandle(modelHandle)`) which use the handles to retrieve CPU resources from the managers, check their versions for changes, and create or update GPU resources as needed before rendering. This flow abstracts away the complexities of GPU resource management while still teaching learners about the underlying concepts through the factory interfaces and version tracking mechanisms. Additionally, the factories provide a centralized location for implementing optimizations such as GPU resource caching, lazy updates, and hot-reloading support, allowing learners to see these concepts in action without needing to implement them from scratch.

In some cases the factory abstraction may not be sufficient for advanced use cases that require more direct control over GPU resources. For example, implementing shadow map arrays for cascaded shadow mapping may require manual configuration of texture arrays, depth formats, and sampler settings that go beyond the capabilities of the current factory methods. In such cases, learners can still access the underlying WebGPU API directly to create custom GPU resources while still benefiting from the educational value of the factories for more common scenarios. The engine’s design allows for this flexibility by providing escape hatches where learners can bypass the factories when necessary, while still encouraging the use of factories for most resource creation tasks to maintain consistency and reduce boilerplate. Such features could also be added directly into the factories as future extensions, but the current implementation focuses on providing a solid foundation for teaching core GPU programming concepts through the factory pattern while allowing for direct API access when needed for more complex scenarios. This approach eases the implementation of the engine itself and gives learners a clear educational path to understand GPU resource management concepts without being overwhelmed by the verbosity of the WebGPU API, while still allowing them to explore the underlying API when they are ready for more advanced topics.

With shader-driven pipeline creation established, and building on the architectural and scene-graph foundations above, the engine implements a complete multi-pass rendering pipeline with physically-based materials that demonstrates modern GPU programming concepts while providing a visually rich experience for learners. The pipeline consists of several distinct render passes executed sequentially each frame, with data dependencies flowing from earlier passes to later ones. [Figure 4.1](#) illustrates the complete rendering

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

pipeline and data flow between passes. The **Shadow Pass** renders the scene from each light's perspective into depth textures (shadow maps), producing directional shadow maps for sun lights, spot shadow maps for focused lights, and cube maps for omnidirectional point lights using PCF (Percentage Closer Filtering) for soft shadow edges. The **Mesh Pass** renders visible geometry using shaders and pipelines derived from each renderable's assigned material, an example of dynamic pipeline selection based on material properties. The default `PBR_Lit_Shader.wgsl` implements physically-based rendering with lighting and shadow sampling from bind group 1, but custom shaders can implement different rendering techniques (unlit, toon shading, custom effects) without requiring lighting or shadow map access. The **Debug Pass** renders debug visualizations (light gizmos, bounding boxes, coordinate axes) directly onto the mesh pass render target without depth testing. The **Composite Pass** combines render targets using a full-screen quad technique, applying tone mapping (HDR to LDR conversion) and optional post-processing effects like bloom or color grading. Finally, the **UI Pass** renders ImGui debug interfaces directly onto the swap chain, overlaying UI elements atop the composited scene. Each pass is implemented as a subclass of `RenderPass` and configures its own pipeline state, bind groups, and render targets, following a modular render pass design with explicit resource dependencies.

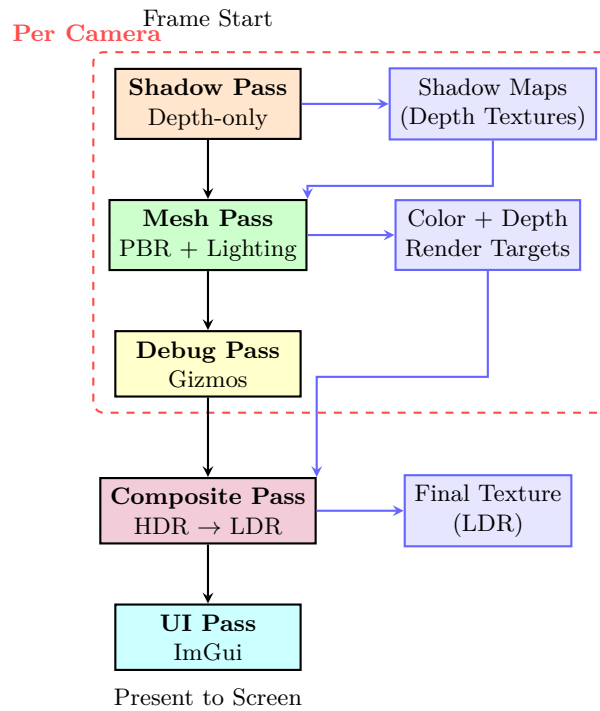


Figure 4.1.: Multi-pass rendering pipeline with per-camera execution (red box) and data dependencies (blue arrows).

The engine allows for different shaders and material properties that can be assigned to a model and its submeshes, which keeps the rendering pipeline flexible. Shaders can

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

be written in WGSL and then registered with the engine’s **ShaderFactory**, as described earlier in this section. By default the engine provides a standard PBR shader `PBR_Lit_Shader.wgsl` that supports the metallic-roughness workflow, but learners can create their own shaders with different BRDFs or material models and assign them to materials. The engine’s material system supports albedo, metallic, roughness, normal maps, and ambient occlusion textures, allowing for a wide range of visual effects. The renderer automatically creates the appropriate pipeline based on the material’s assigned shader and properties, so learners can observe how multiple pipelines and bind groups are managed in a complex rendering system. It also implements the standard light and shadow bind groups of the engine so that models using the PBR Shader can present shadows. Custom shaders can look into the implementation for guidance on how to integrate with the engine’s pipeline and bind group management. As the engine is designed to be extensible, learners can add new render passes, shaders, and material properties as they progress through the tutorials (Section 4.4), allowing them to build upon the core rendering concepts while still benefiting from the automation and structure provided by the engine’s architecture. The multi-pass pipeline manages complex rendering workflows while still providing a clear educational path for learners to understand the underlying GPU programming concepts through practical implementation and visual results.

Figure 4.2 shows the main rendering scene used to demonstrate the engine’s visual capabilities in practice. The two captures show the same content under different post-processing states, highlighting that the pipeline can preserve scene readability while supporting stronger cinematic grading when desired. Having established how scene organization and multi-pass rendering operate at runtime, the next section explains how these systems are delivered pedagogically through the tutorial sequence.

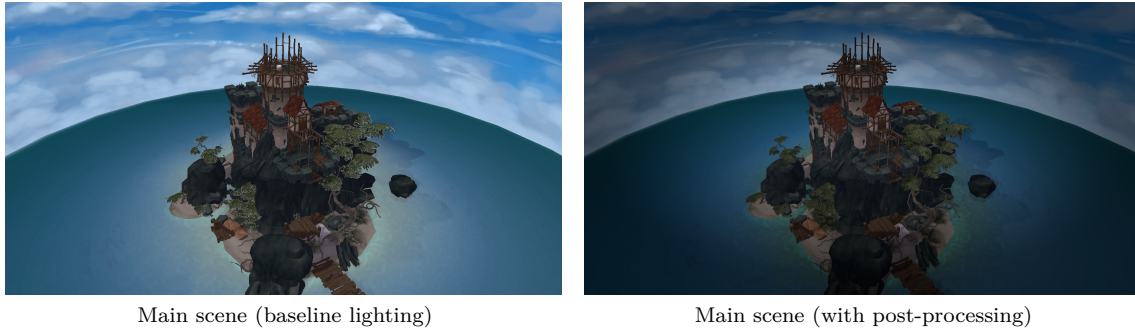


Figure 4.2.: Main rendering scene illustrating the engine’s practical visual output.

Scene attribution (required by license): This work is based on “Sea Keep “Lonely Watcher”” (<https://sketchfab.com/3d-models/sea-keep-lonely-watcher-09a15a0c14cb4acc4060a92bc70413d>) by Artjoms Horosilovs (https://sketchfab.com/Artjoms_Horosilovs), licensed under CC-BY-NC-SA-4.0 (<http://creativecommons.org/licenses/by-nc-sa/4.0/>). License requirements: author credit is mandatory, commercial use is not allowed, and modified versions must be distributed under the same license.

4.4. Tutorial Series and Progressive Learning Design

In order to answer RQ3 regarding the educational effectiveness of the engine, a series of tutorials was designed following the worked-example approach, a learning method established in cognitive science where complete solutions are presented alongside contextual explanations rather than confronting learners with unguided problem-solving tasks [Swe88]. Sweller demonstrated that conventional problem-solving through means-end analysis creates high cognitive load that cannot be utilized for building schemas, the central mechanism for developing expertise. Sweller also showed that guided worked examples prove more effective for novices than free problem-solving. This finding is confirmed by a systematic literature review by Berssanette and de Francisco [BdF22], which analyzed 33 studies on Cognitive Load Theory in programming education and identified the worked-example effect as the most frequently employed and most strongly evidenced concept, particularly relevant since programming is considered a highly cognitively demanding activity requiring simultaneous mastery of multiple competencies. For that reason, the tutorials are structured as step-by-step follow-along implementations, a format popularized by commercial game engine tutorials such as those provided by Unity Technologies, where learners actively reproduce and implement each step rather than passively reading solutions. Each step includes not only the code to be added with clearly marked integration points, but also detailed explanations of what the code does, ensuring that understanding of underlying GPU programming concepts takes priority over mere code transcription.

Given the cognitively demanding nature of GPU programming and the need to manage learner cognitive load effectively, the tutorial series was deliberately limited in scope and duration. Pilot testing conducted during tutorial development revealed that participants strongly prefer tutorial sessions under one hour total duration including questionnaire completion, as longer sessions lead to diminished engagement. This observation aligns with cognitive load research emphasizing that extended learning sessions without adequate breaks exhaust working memory capacity, particularly for complex technical content, and with the author’s own experience completing similar educational studies where extended time commitments create participation barriers. Consequently, Tutorial T03, which would have covered shadow pass implementation and cascaded shadow mapping, was scoped out of the evaluated intervention window to keep session length aligned with these constraints. The resulting tutorial series focuses on three core GPU programming concepts that systematically build learners’ skills.

All tutorials follow a consistent structure and are provided in three formats (Markdown, HTML, PDF) to accommodate different learning preferences. Each tutorial begins with a brief introduction establishing the learning objectives, followed by a step-by-step implementation guide with annotated code snippets and conceptual explanations. The tutorial series implements a corresponding example project within the engine repository where insertion points for learner code are marked with comments (Figure 4.3) such as `// Tutorial 01 - Step 2`. Each tutorial includes clear instructions on where to add code, what the code does, and how it integrates with the existing engine architecture, explicitly referencing the target file and comment marker for each code addition.

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

Step 2: Define Vertex Input Structure

Before we write shaders, we need to understand what data flows through them. Let's start with what the vertex shader receives from the mesh.

Your Task:

Open `examples/tutorial/assets/shaders/unlit.wgsl` and find the comment: `// Tutorial 01 - Step 2`

Add this code:

```
struct VertexInput {
    @location(0) position: vec3f,
    @location(1) normal: vec3f,
    @location(2) texCoord: vec2f,
}
```

What it contains:

- `position` - 3D vertex position in model space (where the vertex is in the model)
- `normal` - Surface normal vector (which way the surface faces, used for lighting)
- `texCoord` - UV coordinates (where on the texture to sample, range 0-1)

Why these attributes? The engine's mesh loader provides these three attributes for every vertex. The `@location(N)` numbers map to the vertex buffer layout defined in C++. While the engine's mesh format includes other properties (tangents, vertex colors), you only need to declare the attributes your shader actually uses. We'll cover shader reflection in detail in Tutorial 03.

Figure 4.3.: Example of a tutorial step showing the code block, contextual explanation, and integration points.

Each tutorial concludes with four standardized sections: *Example Modifications* providing extension exercises for independent practice, *Key Takeaways* summarizing essential concepts, *Further Reading* referencing advanced material, and *Troubleshooting* addressing common implementation issues with solutions. This structure balances guided instruction with opportunities for exploration, reinforcing the worked-example approach while encouraging deeper engagement through optional challenges.

Tutorial T01 introduces WGSL shader programming fundamentals through a minimal unlit shader implementation. Learners receive a pre-configured example project requiring only WGSL shader code additions and a single C++ line to apply the material to the model. This minimal-modification approach provides immediate visual feedback while teaching shader structure, resource bindings (`@group` and `@binding` attributes), and integration with the engine's rendering pipeline, establishing foundational GPU programming concepts without overwhelming cognitive load.

Tutorial T02 advances to custom bind group creation and shader reflection, introducing GPU resource management patterns. Learners implement a custom bind group for per-object data, modify their shader to consume this data, and integrate it into the scene graph by extending `ModelRenderNode` and leveraging the `BindGroupDataProvider` interface within `preRender()` for automatic per-frame updates. This progression emphasizes resource organization automation through the engine's factory pattern and builder API, reinforcing the architectural principles introduced in [Section 4.3](#) and [Section 4.3](#) while teaching practical WebGPU buffer management and bind group construction.

Tutorial T04 completes the series with multi-pass rendering and post-processing tech-

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

niques. It is special in that it targets actual engine-specific files and thus applies changes in the engine itself instead of the tutorial example project. This should motivate learners to engage with the engine codebase directly and understand how the concepts they are learning apply to the engine’s architecture, rather than just following along in a separate example project. This approach encourages deeper engagement with the engine’s design and implementation while still providing a guided learning experience through the tutorial structure. Learners implement a full-screen quad pass that consumes the main render target and applies post-processing effects to a new output texture, covering render target management, command buffer encoding, and the texture sampling patterns essential for modern rendering pipelines.

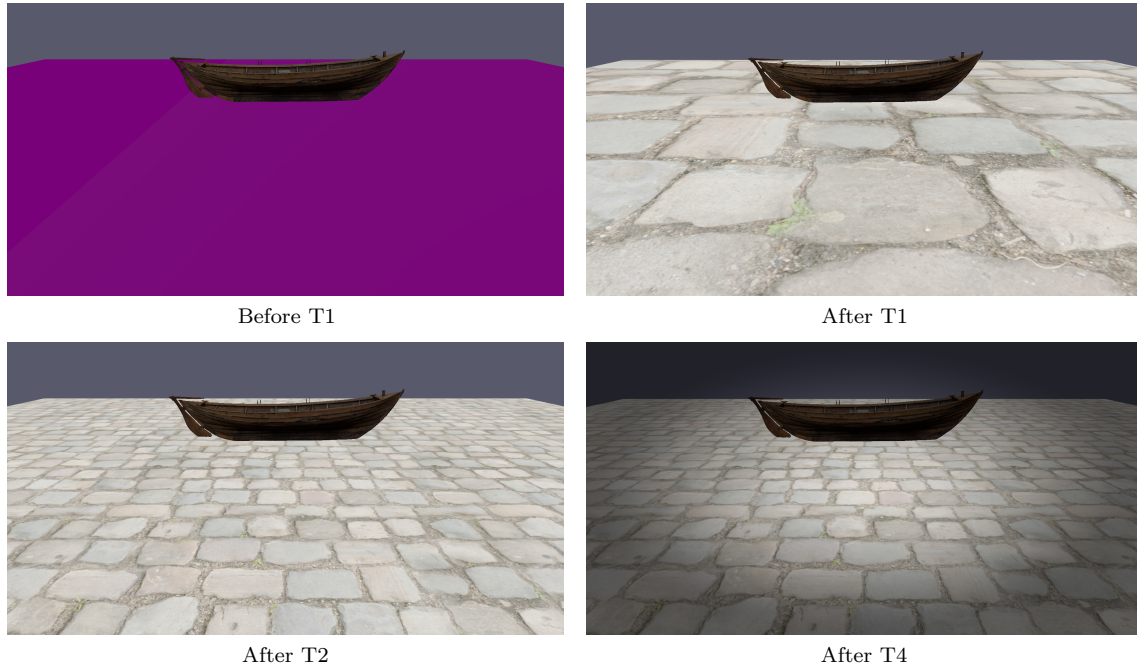


Figure 4.4.: Tutorial progression shown as a 2x2 visual sequence from baseline scene setup to post-processing integration across tutorial milestones.

The tutorial progression balances engine-specific architecture with transferable WebGPU concepts. While each tutorial operates within the engine’s framework, the underlying GPU programming principles, namely buffer creation, bind group layouts, render pass configuration, and command buffer submission, apply directly to raw WebGPU development. By the series conclusion, learners have not only extended the engine with new rendering features but also acquired foundational knowledge of WebGPU resource management, shader development, and multi-pass rendering architectures applicable to independent GPU programming projects beyond the Vienna WebGPU Engine. [Figure 4.4](#) shows the visual progression across the implemented tutorial milestones.

4.5. Implementation Summary and Cross-Platform Considerations

After presenting platform constraints, architectural patterns, runtime rendering, and tutorial delivery, this chapter closes with a summary of the engine’s implementation: a modular architecture that abstracts GPU complexity while exposing essential concepts for educational purposes. The engine implements physically-based rendering with Cook-Torrance BRDF and a metallic-roughness workflow, as well as shadow mapping for directional, spot, and point lights with PCF filtering. The multi-pass rendering architecture executes five distinct passes (shadow, mesh, debug, composite, UI), and the hot-reloadable shader system enables runtime modification without application restart. A Unity-inspired scene graph provides lifecycle methods familiar to learners, while type-safe handle-based resource management prevents entire classes of pointer errors. The factory pattern reduces operation-level WebGPU boilerplate by introducing concepts through concise high-level calls before revealing internal implementation details for extension, and three progressive tutorials teach WGSL shaders, custom bind groups, and multi-pass rendering. The glTF/GLB import path is functional for static meshes and materials, while animation import remains future work. Taken together, these features demonstrate that a fully functional educational rendering engine can be built on WebGPU within the scope of this thesis.

The implementation uses `wgpu-native v0.19.4.1` [gfx26] as the cross-platform WebGPU C API foundation, with Élie Michel’s `webgpu-hpp` wrapper [Mic] providing type-safe C++ bindings. SDL handles windowing and input, ImGui provides debug UI, and GLM supplies graphics-oriented mathematics. Cross-platform testing validated portability across Windows, macOS, and Linux native platforms, with platform-specific problems such as Metal backend bind group ordering strictness and Linux SDL2 Wayland initialization behavior that had to be resolved (detailed in Section 4.1). Near the end of this work the windowing layer was migrated from SDL2 to SDL3. This migration was largely a renaming of the affected calls, and because it came late, the cross-platform validation reported here was carried out on SDL2 and has not yet been repeated on SDL3. Re-running that validation on SDL3 is left to future work. Web deployment remains future work which will be addressed after a migration to the newest version of WebGPU in the Dawn implementation.

Performance was not a primary development focus, though informal observations indicate the engine handles scenes with dozens of models (each containing thousands of triangles), multiple shadow-casting lights, and PBR materials with normal mapping while maintaining consistent framerates on modest hardware (Intel integrated graphics, mid-range discrete GPUs). However, minimal optimization has been performed. The factory caching system and lazy resource updates provide basic efficiency, but advanced techniques such as frustum culling, occlusion queries, instancing, and GPU-driven rendering remain unimplemented. Physics simulation is also not included, which would have a significant impact on performance. These limitations are acceptable for an educational engine where performance optimization would introduce complexity detracting from pedagogical clarity.

Overall this chapter addresses the three research questions via systematic implementa-

4. Vienna WebGPU Engine: Educational Rendering Engine Architecture

tion evidence while focusing on a clear separation of evidence types across chapters. RQ1 (“What is the current state of WebGPU as a graphics API compared to established low-level alternatives?”) is partially addressed through successful implementation of modern rendering features, which confirms WebGPU’s suitability for complex graphics applications. It also identifies practical limitations such as missing shader reflection APIs and platform-specific validation differences. This implementation-side analysis is complemented in [Chapter 5](#) by the canonical WebGPU-versus-Vulkan baseline comparison between hello-triangle references, including lines-of-code and distinct API-call counts as quantitative accessibility indicators. In that evaluation context, this canonical baseline serves primarily as confirmatory quantitative evidence for H1 and RQ2, while contributing secondary interpretive context for RQ1. RQ2 (“Is it feasible to build a fully functional rendering engine using WebGPU?”) receives strong support through the implemented feature checklist, which shows that the engine meets completeness criteria for an educational rendering context within the scope of this thesis. RQ3 (“Can such an engine effectively teach modern GPU programming concepts to students?”) is supported by the architectural design decisions presented in this chapter: the factory pattern enables progressive disclosure by reducing early call-site decision pressure and later exposing internals for extension (see [Appendix A.2](#)), the handle system promotes safe resource management by preventing pointer-related errors, and the tutorial sequence provides structured skill progression grounded in cognitive load theory and the worked-example approach. At the same time, final confirmation of RQ3 requires empirical evidence. Therefore, [Chapter 5](#) presents participant questionnaire results using a four-point Likert-scale instrument together with inferential statistical analysis, providing quantitative support for the pedagogical goals established throughout this implementation chapter.

5. Evaluation

This chapter evaluates all three research questions from [Chapter 1](#). Here another overview of the research questions is provided for context:

RQ1: What is the current state of WebGPU as a graphics API compared to established low-level alternatives?

RQ2: Is it feasible to build a fully functional rendering engine using WebGPU?

RQ3: Can such an engine effectively teach modern GPU programming concepts to students?

RQ1 is addressed through a WebGPU ecosystem assessment, RQ2 is operationalized through H1 and tested via an implementation-complexity comparison between canonical WebGPU and Vulkan hello-triangle references, and RQ3 through a user study that tests H2 and H3. The hypothesis definitions are repeated at the beginning of the sections that test them. All questionnaire items use a four-point Likert scale (Strongly Disagree, Disagree, Agree, Strongly Agree), encoded as values from 1 to 4.

5.1. RQ1: WebGPU Ecosystem Assessment

Within this chapter, RQ1 asks whether WebGPU’s ecosystem maturity is sufficient for reliable educational use. WebGPU reached a substantially more mature state by early 2026 than in its initial rollout phase, primarily because Dawn-based implementations and tooling became significantly more robust and predictable in practice. Browser and platform coverage also improved, with stronger support paths on Firefox and Linux than in earlier project phases, which reduced setup friction for student environments. In parallel, legacy Emscripten integration paths were discontinued in favor of workflows that better align with current Dawn/WebGPU backends. The standards process remained active, but the implementation ecosystem was already sufficiently stable for educational use, particularly for teaching explicit resource and pipeline concepts that map well to modern APIs [W3C23b, Ken22, CBH⁺24]. In this thesis context, this maturity level is important because teaching material and examples must remain reproducible over multiple semesters, and API instability directly harms pedagogical continuity. Implementation work in this thesis ran from early 2025 to February 2026, while the project stayed on a stable WebGPU baseline from that period to keep the engine and tutorials reproducible.

At the same time, ecosystem fragmentation remains relevant to RQ1. Native backends, browser implementations, and documentation depth still vary by platform and release, which means that educators should treat environment setup and troubleshooting as explicit course content rather than implicit prerequisites [KWW23, Ust24, JWB⁺25]. [Table 5.1](#) summarizes the main maturity aspects used for this assessment, including stability indicators, active application evidence, and remaining deployment and portability risks.

5. Evaluation

Table 5.1.: RQ1 evidence overview: WebGPU ecosystem maturity for educational use (early 2025 to February 2026).

Aspect	Evidence (scope: early 2025 to February 2026)	Educational implication
Specification status	WebGPU remained in W3C Candidate Recommendation, while the core <code>webgpu.h</code> C API header was declared stable in Chrome 141 [W3C23b, Chr25c].	Core API semantics are stable enough for reproducible teaching material.
Applied ecosystem activity	Multiple peer-reviewed or public implementations demonstrated practical WebGPU use in visualization, WebGIS, and browser-native engine contexts [KWW23, Ust24, JWB ⁺ 25].	Students can connect tutorials to active real-world usage domains.
Performance relevance	Comparative studies reported meaningful WebGPU benefits over WebGL for several workload classes [CBH ⁺ 24].	Using WebGPU in teaching aligns with contemporary performance-oriented API concepts.
Tooling and backend consistency	Native and browser backends still differed in validation strictness and integration behavior (see Section 4.1).	Courses should include explicit environment setup and backend-specific troubleshooting guidance.
Web deployment path stability	Emscripten integration paths changed during the project period before converging toward Dawn-aligned workflows (Section 4.1).	Native-first deployment remains the lower-risk path for course reliability.

The five aspects of Table 5.1 can be read as two groups. The first three rows collect evidence in favor of educational adoption. The specification row shows that, although WebGPU formally remained at the Candidate Recommendation stage, the core `webgpu.h` C API header was declared stable in Chrome 141, which fixes the API surface that teaching material depends on. The applied-activity and performance rows document that peer-reviewed and public implementations use WebGPU productively in visualization, WebGIS, and browser-native engines, and that comparative studies report measurable benefits over WebGL for several workload classes. Students therefore learn an API with demonstrated practical relevance. The last two rows collect the remaining risks: native and browser backends still differ in validation strictness and integration behavior, and the Emscripten-based web deployment path changed during the project period. For course planning, these two issues translate into practical consequences. They indicate that environment setup and troubleshooting should preferably be treated as explicit course content.

To align RQ1 with its broader scope, Table 5.2 adds a qualitative assessment against established low-level alternatives that go beyond the canonical Vulkan baseline. This is not a second code-level benchmark. Instead, it puts WebGPU into context in education-relevant aspects such as abstraction style, setup burden, portability profile, and teaching reproducibility risk [W3C23b, Khr16, App14, Mic15]. The canonical WebGPU-versus-Vulkan complexity comparison in Section 5.2 remains the primary confirmatory analysis for H1 and therefore mainly addresses RQ2. Within RQ1, it is only used as supporting context for interpreting API accessibility.

5. Evaluation

Table 5.2.: RQ1 qualitative cross-API positioning for educational use.

Aspects	Cross-API observation (WebGPU vs established alternatives)	Educational implication
Abstraction model	WebGPU preserves explicit modern resource and pipeline concepts while constraining some low-level control paths that are fully exposed in Vulkan, Metal, and DirectX 12 [W3C23b, Khr16, App14, Mic15].	Introductory teaching can focus on core GPU concepts before advanced platform-specific complexity.
Setup and tooling burden	Vulkan tutorials typically require substantial initialization scaffolding, and Metal and DirectX 12 depend on platform-specific ecosystems, while WebGPU can reduce entry friction in browser-first or backend-managed setups [Ove23, Ken22].	Lower initial friction improves tutorial throughput, but tooling guidance is still required.
Portability profile	Vulkan targets broad native portability, Metal is Apple-specific, and DirectX 12 is Windows-focused. WebGPU spans browser-native use and evolving native backends, but backend behavior remains non-uniform in practice [KWW23, Ust24, JWB+25].	Course deployments should include explicit environment validation and fallback paths.
Reproducibility risk for teaching	All alternatives face toolchain drift, but WebGPU’s active standardization and backend evolution shift risk toward integration differences rather than conceptual API instability [W3C23b, Chr25c].	Native-first classroom baselines with documented backend constraints remain the most robust strategy.

Table 5.2 compares WebGPU with Vulkan, Metal, and DirectX 12 along four education-relevant aspects. Regarding the abstraction model, WebGPU preserves the explicit resource and pipeline concepts of its native counterparts while hiding some of their most error-prone low-level control paths, which allows introductory courses to start from core concepts before platform-specific details. Regarding setup and tooling, Vulkan demands substantial initialization scaffolding, and Metal and DirectX 12 tie a course to one vendor ecosystem, whereas WebGPU reduces entry friction in browser-first or backend-managed setups. The portability row shows complementary trade-offs: Vulkan offers broad native portability, Metal and DirectX 12 are platform-bound, and WebGPU spans browser and native targets at the cost of non-uniform backend behavior. Finally, the reproducibility row indicates that all alternatives suffer from toolchain drift, but WebGPU shifts this risk away from conceptual API instability toward integration differences, which a course can manage through documented environment constraints. The table thus clarifies why WebGPU is neither a simplified abstraction that hides GPU behavior nor an API that overwhelms beginners with setup detail. It exposes the same architectural reasoning as Vulkan but through a cleaner entry point, which makes it well suited to courses where students must learn real GPU concepts without first fighting through boilerplate.

Overall, the evidence supports RQ1 with a nuanced conclusion. WebGPU is sufficiently mature for reproducible teaching-oriented use within the scope of this thesis, while the cross-API positioning shows why it occupies a practical educational middle ground relative to established low-level alternatives. This conclusion rests on triangulation across the ecosystem maturity evidence (Table 5.1), the qualitative cross-API positioning (Table 5.2), and the canonical WebGPU-versus-Vulkan quantitative baseline reported in Section 5.2. The remaining platform and tooling variability does not change this verdict, but it must

be treated as an explicit course-level constraint when the engine is deployed in teaching.

5.2. RQ2 and H1: Implementation Complexity Comparison

RQ2 asks whether introductory implementation complexity is lower in WebGPU than in Vulkan for the selected canonical baseline pair. This RQ2 claim is evaluated through H1, which provides the operational and testable formulation. Its definition from [Chapter 1](#) is repeated here:

H1 (addressing RQ2): Implementing equivalent rendering tasks requires significantly fewer lines of code and fewer distinct API calls (used as an operational proxy for API concepts) in WebGPU than in Vulkan [[Mic22](#), [Ove23](#)], demonstrating WebGPU’s accessibility advantages.

To operationalize H1 reproducibly, two canonical tutorial references were compared: Élie Michel’s LearnWebGPU-Code `step030-vanilla` snapshot and Alexander Overvoorde’s Vulkan Tutorial implementation `15_hello_triangle.cpp` [[Mic22](#), [Ove23](#)]. Complexity metrics were extracted using two complementary methods: `lizard` [[Yin](#)], a static analyser that counts non-comment lines of code (NLOC) and cyclomatic complexity (CCN) without requiring header resolution, and a dedicated Python script that extracts distinct API call symbols from each implementation unit (see [Appendix A.1](#)).

Table 5.3.: Per-file lizard metrics for the H1 canonical baseline pair.

File	NLOC	Avg. NLOC/fn	Avg. CCN	Function count
LearnWebGPU-Code/main.cpp	235	27.1	2.4	7
LearnWebGPU-Code/webgpu-utils.cpp	59	27.0	4.0	2
LearnWebGPU-Code/webgpu-utils.h	5	None	None	0
WebGPU total	299	27.1	2.8	9
VulkanTutorial/15_hello_triangle.cpp	700	17.4	2.8	37

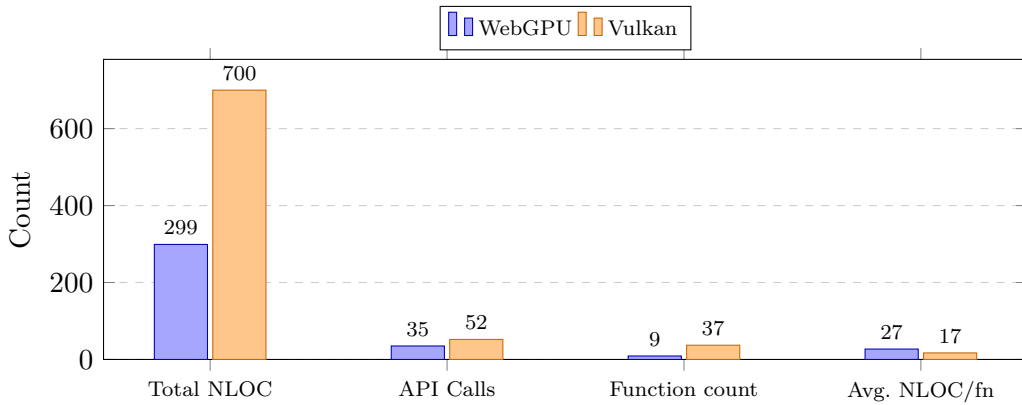


Figure 5.1.: H1 complexity metrics for the canonical baseline pair.

5. Evaluation

For both references, the non-empty non-comment lines of code (NLOC) as well as distinct API call symbols were counted. For the API call counts, the custom script was used alongside lizard’s per-function metrics. For Vulkan, the implementation unit is the single tutorial file, while for WebGPU it is the main runtime file (`main.cpp`) plus the two required synchronous request helper implementations (`requestAdapterSync` and `requestDeviceSync`) in `webgpu-utils.cpp`, which are necessary to achieve an initialisation flow comparable to the Vulkan reference. The `webgpu-utils.cpp` further included optional inspection and reporting helpers, which are not essential for the core hello-triangle logic and therefore were not counted as API call contributors. The header `webgpu-utils.h` contributes five NLOC of declarations and no analysable function bodies, and is listed in [Table 5.3](#) for completeness only. Reproducibility artifacts for this comparison and the RQ3 analysis are listed in [Appendix A.1](#).

[Table 5.3](#) and [Figure 5.1](#) reveal two distinct patterns. First, the Vulkan implementation requires $2.34\times$ more non-comment code than the WebGPU equivalent (700 vs. 299 NLOC) and $1.49\times$ more distinct API call symbols (52 vs. 35), while decomposing the work into $4.11\times$ more functions (37 vs. 9). This directly supports H1 as achieving an equivalent hello-triangle outcome in Vulkan demands substantially more implementation surface across every measured dimension. Second, the average cyclomatic complexity per function is identical for both APIs at 2.8, indicating that the additional Vulkan code does not arise from more intricate branching logic but rather from a larger number of mandatory setup steps, each expressed as a dedicated method. The higher average NLOC per function in the WebGPU case (27.1 vs. 17.4) is consistent with this reading: WebGPU consolidates equivalent work into fewer, somewhat larger functions rather than distributing it across many small ones. Taken together, the metrics characterise Vulkan’s greater complexity as *structural* rather than *algorithmic* in nature.

Overall the comparison supports H1: in this canonical hello-triangle baseline, the WebGPU reference is substantially smaller in both LOC and API-call diversity ([Table 5.3](#), [Figure 5.1](#)). This result does not imply that Vulkan is inferior overall. Rather, it supports the educational claim that WebGPU can reduce initial setup complexity while still exposing explicit modern graphics concepts. The inference is intentionally scoped to this canonical baseline pair and should not be interpreted as a universal complexity ranking across all engine architectures. In summary, RQ2 and H1 are supported for the selected canonical pair: WebGPU achieves the same introductory rendering outcome with substantially lower structural setup overhead than Vulkan.

5.3. RQ3: Educational Effectiveness User Study

RQ3 asks whether the tutorial-based use of the Vienna WebGPU Engine yields measurable educational effectiveness in self-assessed learning and perceived usefulness. It is evaluated through H2 and H3, whose definitions from [Chapter 1](#) are repeated here:

H2 (addressing RQ3): Students report a statistically significant positive pre/post change in self-assessed GPU programming confidence after complet-

5. Evaluation

ing the Vienna WebGPU Engine tutorials. The primary confirmatory test is a one-sided paired Wilcoxon signed-rank test on Q01 to Q25 ($H_1 : \text{POST} > \text{PRE}$, $\alpha = 0.05$), while effect magnitude is reported with rank-biserial correlation and Cohen’s d_z , and uncertainty of the mean paired difference is reported using a 95% confidence interval from the paired t -test framework.

H3 (addressing RQ3): Students perceive the Vienna WebGPU Engine and its accompanying tutorials as an effective and approachable tool for learning modern GPU programming concepts. For Q26 to Q33, agreement is operationalized as responses 3 – 4 (Agree/Strongly Agree) and tested against $p_0 = 0.5$ with a one-sided exact binomial test. Effect strength is interpreted using observed agreement proportions and exact 95% confidence intervals.

The evaluation combines quantitative questionnaire data with informal qualitative feedback, and each instrument is aligned with the research questions rather than chosen for statistical convenience. Quantitative items provide comparable data across participants, while verbal feedback collected after each session helps explain why certain response patterns appear. All questionnaire mappings, analysis scripts, and derived summary tables are published in [Appendix A.1](#) so that the numerical claims remain traceable to explicit artifacts rather than manually assembled spreadsheets.

The study followed a single-group pre/post design with $n = 15$ participants. Before the tutorials, participants completed the baseline block Q01 to Q25. After the tutorial sequence, participants completed the same Q01 to Q25 block and the post-only usefulness block Q26 to Q33. The central metric for H2 is the per-participant composite increase across Q01 to Q25, $\Delta = \overline{\text{POST}} - \overline{\text{PRE}}$. This design directly operationalizes the self-assessed confidence and conceptual understanding gains stated in H2. Tutorial execution environments in this sample included Windows ($n = 11$), Linux ($n = 3$), and macOS ($n = 1$), indicating that the evaluated learning workflow was not restricted to a single desktop platform.

[Table 5.5](#) summarizes participant background and tutorial completion in one overview, where the count column reports each subgroup’s size relative to the full sample and the mean-improvement column denotes the average Q01 to Q25 pre/post gain within that subgroup. Most participants were from computer science programs, with a smaller share from adjacent domains. Tutorial 1 was completed by all participants, while later tutorials had small non-completer groups.

Because non-completer subgroups can be very small, these completion-specific means are reported for context and should not be interpreted as stable subgroup effects. In particular, the slightly higher mean gain in the Tutorial 2 non-completer row (0.926 vs. 0.870) is descriptive only and should not be interpreted causally as a benefit of non-completion. With small, non-random subgroups, such a pattern can arise from baseline differences, for example if some non-completers already started with stronger prior understanding. [Table 5.4](#) provides the questionnaire items used for traceable reporting of all quantitative results in this chapter. It lists every item with its identifier, a shortened wording, its availability (pre and post versus post-only), and the topic group used for the aggregated

5. Evaluation

analyses below. The instrument targets specific constructs rather than broad judgments that mix several factors, with several items per construct to improve reliability and wording kept consistent with the participants' background. The four-point Likert responses are treated as ordinal during analysis.

Table 5.4.: Questionnaire items (Q01 to Q33) with availability and topic mapping.

ID	Question (shortened)	Availability	Topic
Q01	CPU and GPU responsibilities in rendering	BOTH	general
Q02	Basic real-time rendering pipeline	BOTH	general
Q03	Transformations (model, view, projection)	BOTH	general
Q04	Lighting models (Phong, Blinn-Phong)	BOTH	general
Q05	Purpose of physically based rendering	BOTH	general
Q06	Prior shader programming experience	BOTH	shader programming
Q07	Difference between vertex and fragment shaders	BOTH	shader programming
Q08	How CPU data is passed to shaders	BOTH	shader programming
Q09	Texture sampling in shaders	BOTH	shader programming
Q10	Uniform data access in shaders	BOTH	shader programming
Q11	Render pipeline object concept	BOTH	modern GPU API concepts
Q12	Purpose of resource binding	BOTH	modern GPU API concepts
Q13	Uniform buffer concept	BOTH	modern GPU API concepts
Q14	Command buffer usage in modern APIs	BOTH	modern GPU API concepts
Q15	Render pass concept	BOTH	modern GPU API concepts
Q16	Draw call issuance in a GPU API	BOTH	modern GPU API concepts
Q17	Familiarity with the WebGPU API	BOTH	WebGPU specifics
Q18	Purpose of a bind group in WebGPU	BOTH	WebGPU specifics
Q19	Command encoder in WebGPU	BOTH	WebGPU specifics
Q20	Render pass encoder in WebGPU	BOTH	WebGPU specifics
Q21	Pipeline layout and bind group relation	BOTH	WebGPU specifics
Q22	Role of GPU buffers in WebGPU	BOTH	WebGPU specifics
Q23	Confidence in modifying shader code	BOTH	confidence
Q24	Confidence with a modern graphics API	BOTH	confidence
Q25	Confidence in debugging rendering issues	BOTH	confidence
Q26	Tutorials improved general rendering understanding	POST	perceived effects
Q27	Tutorials improved understanding of modern GPU APIs	POST	perceived effects
Q28	Tutorials improved understanding of WebGPU specifically	POST	perceived effects
Q29	Hands-on implementation improved pipeline understanding	POST	perceived effects
Q30	Engine structure made GPU concepts easier	POST	perceived effects
Q31	Comfort with WebGPU increased after tutorials	POST	perceived effects
Q32	Capability to implement a simple WebGPU feature	POST	perceived effects
Q33	Would recommend tutorials to other learners	POST	perceived effects

Block	Category	Count (/15)	Mean improvement
Participant profile (study field distribution)			
Study field	Computer Science	11/15	0.851
Study field	Media Informatics	2/15	0.940
Study field	Cyber Security	1/15	1.240
Study field	Medical Computer Science	1/15	0.960
Tutorial completion and associated mean Q01 to Q25 gain			
Completion	Tutorial 1: Unlit Shader (completed)	15/15	0.896
Completion	Tutorial 2: Custom Bindgroup (completed)	8/15	0.870
Completion	Tutorial 2: Custom Bindgroup (not completed)	7/15	0.926
Completion	Tutorial 4: Post Processing (completed)	14/15	0.909
Completion	Tutorial 4: Post Processing (not completed)	1/15	0.720

Table 5.5.: Participant profile and tutorial completion overview ($n = 15$).

5. Evaluation

The quantitative analysis is organized to separate confirmatory inference from descriptive profiling. Confirmatory tests evaluate whether the directional hypotheses formulated in [Chapter 1](#) are statistically supported under the predefined test strategy, whereas descriptive tables and figures are used to localize where gains or weaker responses occur across topics and individual items. This separation is methodologically relevant because statistical significance alone does not indicate practical magnitude, and descriptive shifts alone do not establish inferential reliability. Therefore, each hypothesis is interpreted using a combined view of test statistics, confidence intervals, and effect sizes, followed by item- and topic-level patterns that provide pedagogical interpretability.

For H2, inferential analysis is based on paired participant-level composite gains over Q01 to Q25. For each participant, PRE and POST responses are averaged across these 25 items, and the participant-level improvement is computed as $\Delta = \overline{\text{POST}} - \overline{\text{PRE}}$. The primary confirmatory test follows the hypothesis statement in [Chapter 1](#): one-sided Wilcoxon signed-rank test ($H_1 : \Delta > 0, \alpha = 0.05$). A paired t -test is reported as a complementary parametric robustness check, and the 95% confidence interval for the mean paired difference is reported from the paired t -test framework. Effect magnitude is reported using rank-biserial correlation and Cohen’s d_z . For H3, post-only agreement rates are evaluated with one-sided exact binomial tests against a neutral baseline of $p_0 = 0.5$.

For H3, responses are dichotomized into agreement (≥ 3) versus non-agreement (< 3) to match the directional hypothesis that a majority of participants evaluates the tutorials positively. The neutral baseline $p_0 = 0.5$ represents an even split between agreement and non-agreement (no majority agreement). In addition to one-sided hypothesis tests, two-sided 95% confidence intervals are reported for key aggregate estimates to quantify uncertainty independently of test direction. Because the eight item-level agreement hypotheses are tested simultaneously, their p -values are adjusted with a Holm correction to control the family-wise error rate.

To frame interpretation, three validity dimensions are considered. Internal validity is supported by the paired pre/post structure, where each participant serves as their own baseline, but cannot fully exclude confounds such as concurrent learning activities outside the study sessions. Construct validity is addressed by mapping items to explicit constructs (general rendering, modern GPU concepts, WebGPU specifics, and confidence) and by reporting block-level reliability. However, all outcomes remain self-assessed. Self-assessment captures perceived understanding and is informative for usability and motivation, but it can diverge from objective task performance, so the results are not treated as direct performance tests. A stronger design would complement the questionnaire with behavioral evidence such as implementation tasks or transfer exercises, which is especially relevant in graphics education where conceptual understanding and practical debugging skill both matter. The validity of the study is limited because the sample is small and comes from a single academic context. Therefore, caution is advised when generalizing these results to other groups. The test strategy combines non-parametric and parametric analyses plus effect sizes to reduce dependence on one modeling assumption and to separate statistical significance from practical relevance.

5. Evaluation

Table 5.6.: Internal consistency of questionnaire blocks (Cronbach’s α).

Block	Participants	Items	Cronbach’s α
Q01 to Q25 POST	15	25	0.949
Q26 to Q33 POST	15	8	0.885

Before interpreting hypothesis tests, internal consistency of the questionnaire blocks is checked. As shown in Table 5.6, both blocks are high in this sample ($\alpha \rightarrow 1$ indicates stronger internal consistency), which supports interpreting composite and aggregated block-level statistics.

The H2 analysis intentionally proceeds from global composite change to progressively finer granularity. First, the chapter establishes whether there is reliable overall movement across the full Q01 to Q25 block. Second, it identifies which conceptual clusters contribute most to that movement. Third, it inspects participant-item structure to distinguish broad improvement from isolated outliers. Interpreting these levels jointly is important in a small-sample educational study, because aggregate significance may coexist with heterogeneous item behavior. The following tables and figures should therefore be read as complementary layers of evidence rather than independent claims.

Descriptive composite statistics are listed in Table 5.7. The mean score increased from 1.971 (PRE) to 2.867 (POST), corresponding to an average gain of 0.896 points on the 4-point scale. Descriptively, this is an increase of about 45.5% relative to the PRE mean.

Table 5.7.: Composite pre/post statistics for Q01 to Q25 ($n = 15$).

Timepoint	Mean	Median	SD	Count
PRE	1.971	1.960	0.657	15
POST	2.867	2.880	0.597	15

The overall trend in Table 5.7 suggests that the improvement is not driven solely by a few individuals with particularly large gains. The mean and median shift in parallel from PRE to POST, which is consistent with a broadly distributed upward trend among participants rather than a highly skewed change profile. The slight decrease in the standard deviation from PRE to POST indicates a moderate convergence of self-assessments following the tutorial series. This convergence is pedagogically relevant, as it suggests that participants not only improved on average but also converged in their perceived understanding of the concepts covered.

Table 5.8.: H2 inferential summary for participant-level Q01 to Q25 gains ($n = 15$).

Measure	Value	Role in interpretation
Wilcoxon signed-rank (one-sided)	$W = 120, p = 0.000323$	Primary confirmatory test for H2
Paired t -test (one-sided)	$t(14) = 11.150, p < 0.001$	Parametric robustness check
Mean paired gain	0.896	Magnitude on 4-point scale
95% CI for mean paired gain	[0.724, 1.068]	Uncertainty of mean change
Effect sizes	$d_z = 2.879, r_{rb} = 1.000$	Practical relevance

5. Evaluation

Inferential results for H2 are summarized in Table 5.8, which reports the primary confirmatory test together with the parametric robustness check, the mean paired gain, its 95% confidence interval, and both effect size measures. As the table shows, the primary Wilcoxon signed-rank test supports H2 at $\alpha = 0.05$. The one-sided p -value is derived from the 15 paired participant-level Δ values under the directional null hypothesis that median improvement is not greater than zero, so the result provides strong evidence for positive median change. The paired t -test points in the same direction.

Taken together, the inferential results suggest both statistical robustness and practical relevance. The convergence of the non-parametric and parametric tests reduces dependence on one distributional assumption in a modest sample. The confidence interval is entirely above zero and remains sizable relative to the 4-point scale, so the observed gain is not only statistically detectable but also large enough to matter for an educational intervention of this duration.

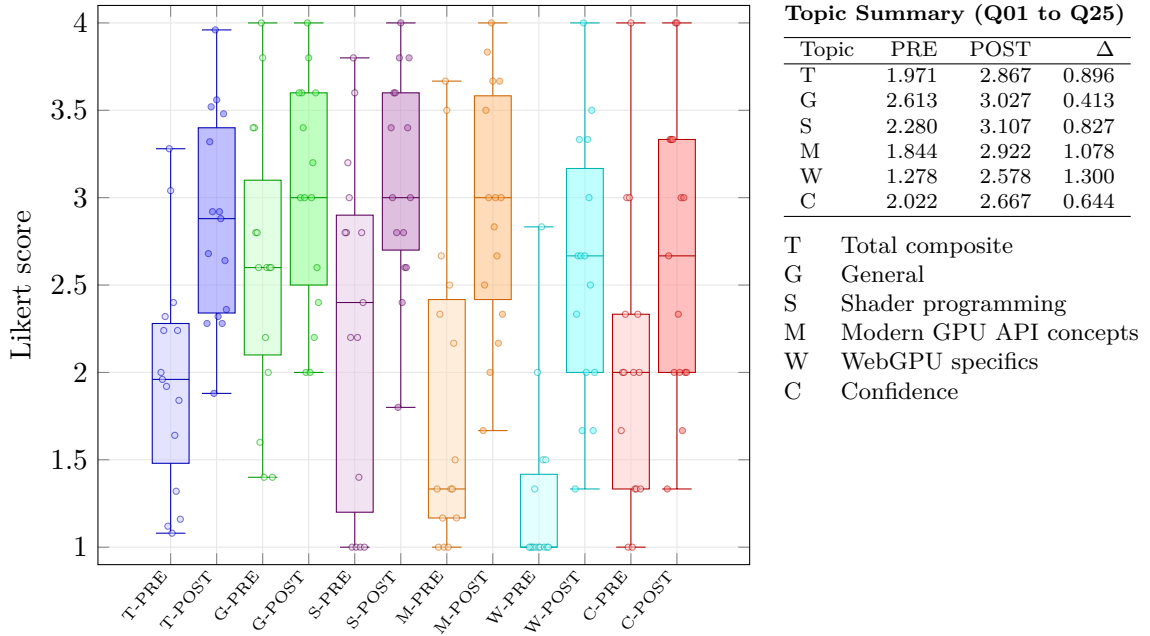


Figure 5.2.: Combined topic-level H2 evidence.

Figure 5.2 combines topic-level numeric summaries (right panel) with PRE/POST distributional evidence and participant-level points (left panel). The largest average increase appears for WebGPU specifics (+1.300), followed by modern GPU API concepts (+1.078). This combined view supports two linked interpretations. First, the intervention appears strongest where baseline familiarity was lowest: WebGPU-specific and modern API concept blocks exhibit the largest absolute gains, which aligns with the tutorial focus on explicit resource binding, command encoding, and pipeline reasoning. Second, the distributional panel shows that upward movement occurs across all topic groups, not only in one domain. In most groups, POST boxes are shifted upward relative to PRE and

5. Evaluation

participant-level points show broadly positive displacement, indicating cohort-wide progress rather than isolated outlier-driven effects. The smaller increase in “general” and confidence-related blocks remains plausible under partial ceiling effects and different rates of conceptual versus self-efficacy change.

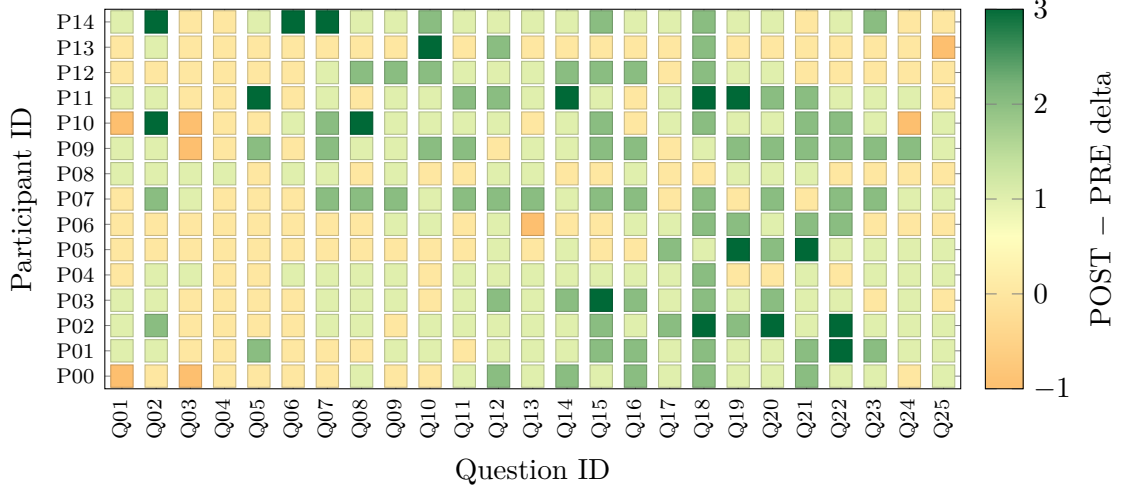


Figure 5.3.: Participant-level heatmap of Q01 to Q25 deltas (POST – PRE). Greener colors indicate larger positive gains. More reddish color tones indicate decreases.

Figure 5.3 visualizes participant-level deltas for all 25 pre/post items. Most cells are non-negative, and high-gain regions cluster around WebGPU-specific and modern API concept items, which is consistent with the topic-level pattern shown in Figure 5.2. The heatmap perspective adds two exploratory insights. First, gains are not confined to one or two participants. Positive deltas appear across many rows, indicating distributed learning effects throughout the sample. Second, larger gains concentrate in the same item families highlighted by the topic table, providing cross-level consistency between aggregate and participant-item evidence. Where negative cells occur, they are sparse and low magnitude relative to the dominant positive pattern, so they appear as local recalibrations rather than a competing global trend.

Although most deltas are non-negative, a small number of item-level decreases were observed (8 of 375 participant-item entries). This pattern is plausible for self-assessment tools: after reviewing the material, some participants reassess their own competence more critically, rather than holding on to their initial overconfidence. Informal verbal feedback obtained immediately after completing the questionnaire was consistent with this interpretation.

The strongest item-level increases were observed for Q18 (bind groups in WebGPU, +1.867), Q15 (render pass concept, +1.333), and Q20 (render pass encoder in WebGPU, +1.333). In contrast, foundational topics with higher initial familiarity, such as Q03 (transformations), showed little to no change.

5. Evaluation

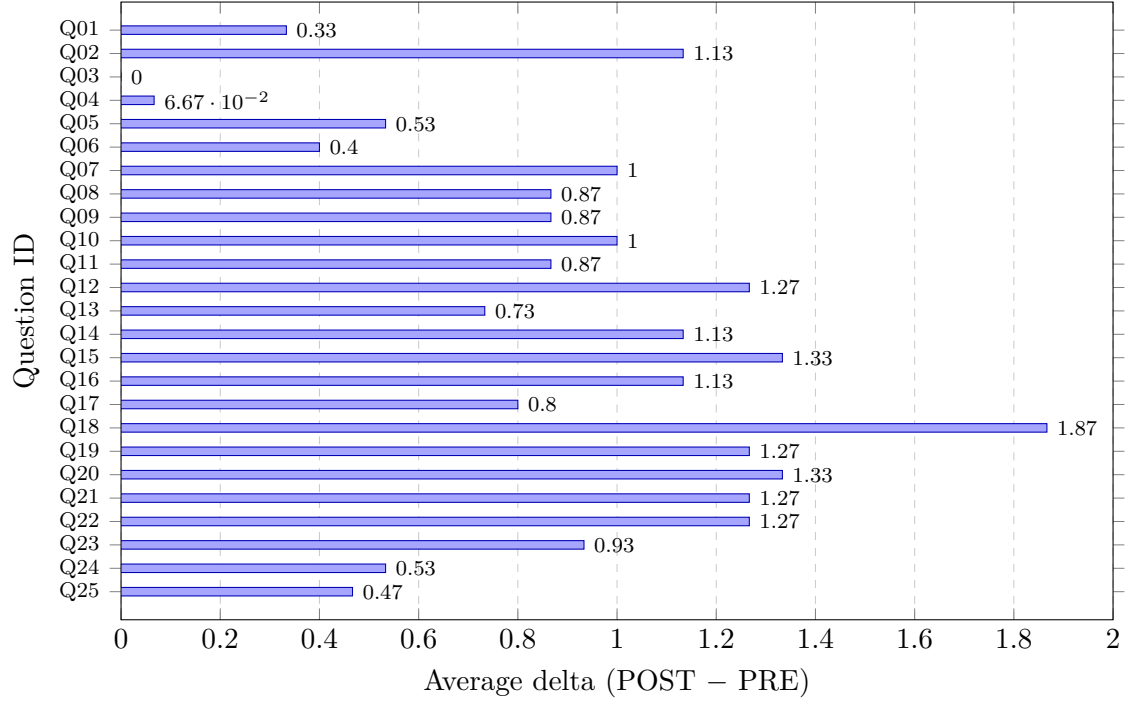


Figure 5.4.: Mean score increase per item across Q01 to Q25. High-growth items align with core tutorial constructs.

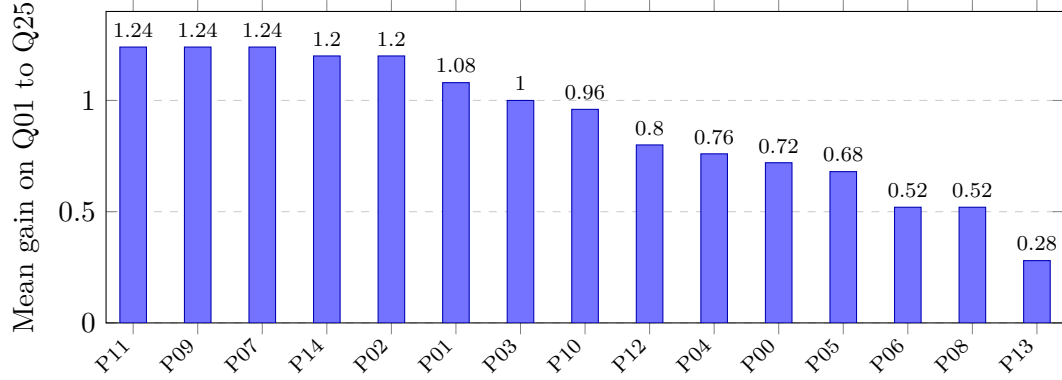


Figure 5.5.: Participant-level ranking by mean gain across Q01 to Q25.

Figure 5.4 shows the mean score increase per item across Q01 to Q25, making visible that high-growth items align with core tutorial constructs. Figure 5.5 shows that participant means are positive across the full ranking, indicating that the learning gain is consistent rather than driven by a subset of high-performing participants. This combination reduces the risk of over-interpreting single-item peaks because participant-level averages still indicate broad improvement. At the same time, the spread in participant

5. Evaluation

means confirms heterogeneity in gain magnitude, which motivates the later subgroup and limitations discussion rather than implying uniform effect sizes for all learners.

As an exploratory subgroup check using the user-selected study-level grouping (Master/PhD versus Others), the gain difference was small and not statistically significant ($U = 27$, $p = 1.000$, two-sided Mann-Whitney test). Group means were 0.916 (Master/PhD, $n = 9$) and 0.867 (Others, $n = 6$), suggesting similar learning gains across study-level groups in this sample. In summary, H2 shows a strong overall positive pre/post shift, while subgroup patterns remain exploratory and are best interpreted as context signals rather than causal contrasts.

Figure 5.6 summarizes the H3 item profile in one place. The top panel shows agreement rates (responses ≥ 3), which range from 60.0% (Q28) to 86.7% (Q26, Q33). The bottom panel shows the full four-category Likert composition per item.

Aggregating all eight items yields 89/120 agreement responses (74.2%), which is significantly above the 50% baseline. Here, 120 is the total number of post-only H3 responses (15 participants $\times$ 8 items). The exact 95% CI is [65.4%, 81.7%], and the one-sided exact binomial test against $p_0 = 0.5$ yields $p = 5.59 \times 10^{-8}$ (thus also $p < 0.001$). The aggregate mean score across Q26 to Q33 is 3.00 on the 4-point Likert scale. Relative to the 50% benchmark, this corresponds to a +24.2 percentage-point agreement margin. This supports H3 at the aggregate level. Item-level exact binomial tests with Holm correction are summarized in Table 5.9. After correction, Q26 and Q33 remain significant. The table also explains why strong aggregate support does not translate into uniform item-level significance. Several items such as Q29 and Q30 reach raw significance but rise above the threshold after Holm correction, which is expected with only 15 responses per item. The two items that survive correction, Q26 and Q33, also carry the highest agreement counts (13 of 15 each), so the strongest item-level evidence concerns general rendering understanding and willingness to recommend the tutorials. The combined view confirms that positive categories (Agree and Strongly Agree) dominate most items, while Q28 and Q31 retain comparatively larger disagreement shares and therefore remain natural targets for tutorial refinement.

Table 5.9.: H3 item-level agreement tests (one-sided exact binomial, Holm-adjusted).

Item	Question (shortened)	Agreements	Total	p_{raw}	p_{Holm}
Q26	Tutorials improved general rendering understanding	13	15	0.0037	0.0295
Q27	Tutorials improved understanding of modern GPU APIs	10	15	0.1509	0.6035
Q28	Tutorials improved understanding of WebGPU specifically	9	15	0.3036	0.6035
Q29	Hands-on implementation improved pipeline understanding	12	15	0.0176	0.1055
Q30	Engine structure made GPU concepts easier	12	15	0.0176	0.1055
Q31	Comfort with WebGPU increased after tutorials	10	15	0.1509	0.6035
Q32	Capability to implement a simple WebGPU feature	10	15	0.1509	0.6035
Q33	Would recommend tutorials to other learners	13	15	0.0037	0.0295

5. Evaluation

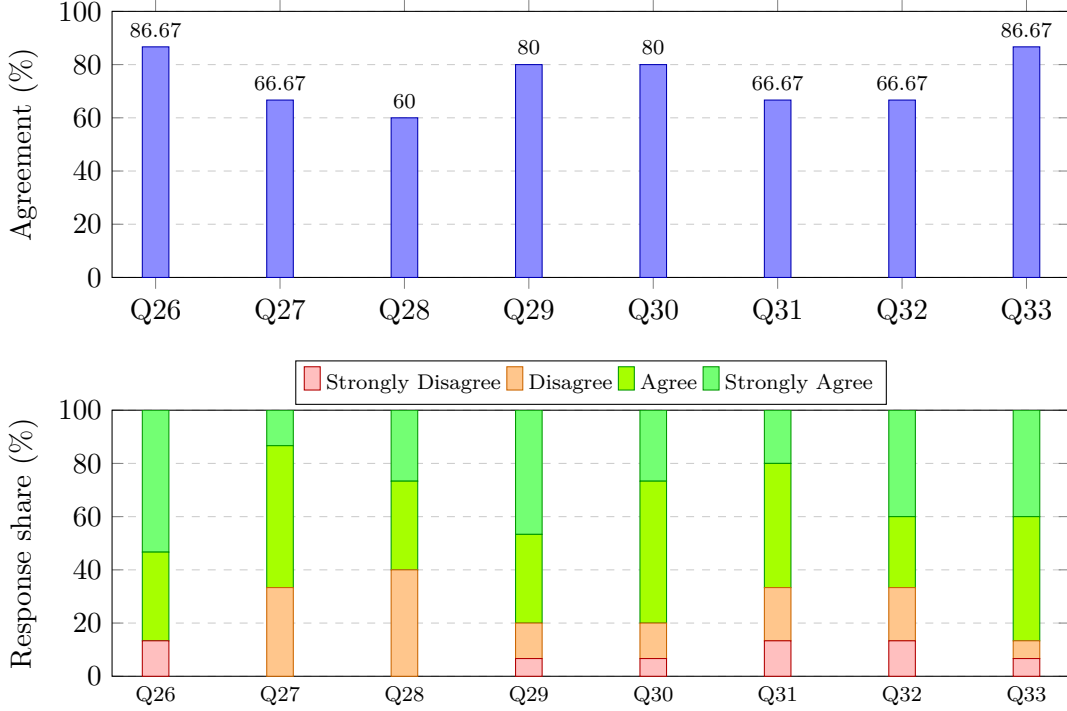


Figure 5.6.: Combined H3 item view: top panel shows agreement rates (responses ≥ 3), bottom panel shows stacked Likert distributions for Q26 to Q33 (post-only block).

Exploratory study-level subgroup analysis for H3 (Master/PhD versus Others) showed agreement rates of 80.6% (58/72) and 64.6% (31/48), respectively. A two-sided two-proportion comparison yielded $z = 1.958$ and $p = 0.050$, indicating a borderline pattern that should be interpreted cautiously due to the small sample sizes and exploratory nature of subgroup analysis. In summary, H3 is supported at aggregate level, with clearly positive overall perceived usefulness and heterogeneous but directionally favorable item-level agreement profiles. Overall, RQ3 is supported in this sample: the tutorial intervention is associated with substantial self-assessed learning gains and positive perceived educational effectiveness, while subgroup detail remains exploratory.

5.4. Discussion

The evaluation provides converging, but not uniform, evidence across all three research questions. For RQ1, the combined implementation and user-study results indicate that WebGPU is viable for teaching modern graphics concepts when supported by strong abstractions and structured learning materials. At the same time, this conclusion depends on the current toolchain context and should not be interpreted as a claim that WebGPU workflows are frictionless in all environments.

5. Evaluation

For RQ2, the complexity comparison supports the thesis claim at the intended level: the selected educational baseline suggests substantially lower implementation overhead than Vulkan for the same introductory rendering goals. This result should be read as evidence of pedagogical feasibility rather than as a universal benchmark against production-grade Vulkan engines. The implemented feature set was intentionally scoped toward conceptual coverage, not exhaustive engine completeness, and the comparison therefore reflects teaching-oriented priorities.

For RQ3, the strongest quantitative signal appears in H2. Pre/post results show statistically significant and practically large gains in self-assessed understanding, especially for concepts that are central to explicit GPU APIs such as resource binding and command encoding. This alignment between measured gains and targeted architecture design is consistent with the didactic intent of the factory-based workflow. H3 is also supported at aggregate level, but item-level results are more heterogeneous after correction for multiple testing, which is expected under small per-item sample sizes and conservative family-wise procedures.

Several limitations constrain interpretation. The sample is modest and drawn from a single academic context, which limits external validity. Core outcome measures are self-reported and therefore susceptible to response bias. Subgroup analyses, especially tutorial-completion subsets, remain exploratory because of limited statistical power. In addition, tutorial-based learning can drift into copy-first behavior if reflective tasks are too weak. Future iterations should therefore strengthen active learning with required variations, short comprehension checkpoints, and assessment criteria that reward explanation and transfer, not only successful replication.

Overall, the chapter supports a calibrated conclusion: WebGPU can serve as an effective educational foundation when paired with architecture and pedagogy designed for learnability, and the Vienna WebGPU Engine provides one concrete realization of this approach. The findings are strong enough to motivate broader adoption and follow-up studies, while still requiring replication across larger and more diverse cohorts to establish stronger generalizability.

6. Conclusion

Modern graphics programming education still faces a fundamental trade-off: APIs that expose state-of-the-art GPU concepts often introduce substantial setup complexity, while highly abstracted engines can hide the very concepts students need to learn. This thesis addressed that gap by investigating WebGPU as a potential middle ground for teaching explicit modern GPU programming with lower entry friction than traditional low-level APIs.

To answer this challenge, the thesis designed and implemented the Vienna WebGPU Engine as an educational rendering engine in C++ using WebGPU. The implementation focused on architectural consistency and teachability through a layered design, factory-based GPU resource creation, type-safe resource handles, a Unity-inspired scene graph, and a multi-pass rendering pipeline. In parallel, the thesis developed a tutorial sequence that introduces shader programming, bind group management, and post-processing through guided, hands-on steps.

The findings across the three research questions are convergent. For RQ1, the ecosystem assessment shows that WebGPU reached a practically usable level for teaching by the end of the project period (early 2025 to February 2026), while still exhibiting backend and platform fragmentation that educators must plan for. For RQ2, the implementation demonstrates that a fully functional educational engine on WebGPU is feasible, including PBR shading, multi-light workflows, and shadowing. For RQ3, the user study shows substantial self-reported learning gains and clearly positive perceived educational value (see [Section 5.1](#), [Section 5.2](#), and [Section 5.3](#)).

The hypothesis outcomes are consistent across all three tests. H1 is supported in the canonical hello-triangle baseline: WebGPU required 299 LOC versus 700 LOC for Vulkan (57.3% fewer lines) and 35 versus 52 distinct API calls (32.7% fewer calls), as shown in [Figure 5.1](#). This canonical complexity baseline is interpreted primarily as confirmatory evidence for H1/RQ2 and secondarily as supporting accessibility context for RQ1. H2 is supported by the one-sided Wilcoxon signed-rank test ($p = 0.000323$). The mean paired gain was 0.896 points with a 95% CI of [0.724, 1.068], indicating a large improvement in this sample ([Section 5.3](#)). H3 is also supported at aggregate level: 89 of 120 responses were agreement responses (74.2%), with exact 95% CI [65.4%, 81.7%]. Against the no-majority-agreement benchmark ($p_0 = 0.5$), the one-sided exact binomial test gives $p = 5.59 \times 10^{-8}$ (therefore also $p < 0.001$), which corresponds to a +24.2 percentage-point margin above the benchmark. After Holm correction, item-level significance remains concentrated in the strongest global-usefulness items ([Section 5.3](#), [Table 5.9](#)).

These conclusions should be interpreted within clear scope limits. During the thesis implementation window, development and most native validation were centered on Windows using `wgpu-native` v0.19.4.1, while user-study tutorial execution also succeeded on Linux ($n = 3$) and macOS ($n = 1$). Because these non-Windows subsamples are small, cross-platform conclusions should still be interpreted as preliminary. Web deployment

6. Conclusion

was deferred because of instability and transitions in Emscripten WebGPU integration paths during the project period, and the engine therefore remained on a stable WebGPU baseline from that time. In addition, the user study sample was modest and drawn from one academic context, and the main outcome variables are self-assessed rather than direct performance measures.

Future work can strengthen both technical and pedagogical impact. On the technical side, the engine should migrate to emdawnwebgpu-based web deployment paths, broaden validation depth and packaging robustness for Linux and macOS, re-run the cross-platform validation on the SDL3 windowing layer adopted at the end of this work, and continue reducing platform-specific integration friction. On the pedagogical side, the tutorial series should be expanded with additional advanced units and stronger anti-copy scaffolding through guided modification tasks and short comprehension checks. Larger multi-cohort studies can then evaluate transfer performance beyond self-reported confidence and understanding.

Overall, the Vienna WebGPU Engine demonstrates that WebGPU can serve as a practical foundation for modern GPU programming education. The project contributes both an implementable architecture and empirical evidence that structured, explicit WebGPU workflows can make advanced rendering concepts more accessible to learners without abandoning core low-level principles.

Bibliography

- [AMHH18] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. CRC Press, 4th edition, 2018.
- [App14] Apple Inc. Metal graphics api, 2014. Accessed: 2025-01-30.
- [Bab] Babylon.js Team. Babylon.js webgpu backend. Accessed: 2025-01-30.
- [BdF22] João Henrique Berssanette and Antonio Carlos de Francisco. Cognitive load theory in the context of teaching and learning computer programming: A systematic literature review. *IEEE Transactions on Education*, 65(3):440–449, 2022.
- [Bev] Bevy Engine Team. Bevy engine wgpu backend. Accessed: 2025-01-30.
- [Bil02] Scott Bilas. A data-driven game object system. In *Game Developers Conference (GDC)*, 2002. GDC Vault. Accessed: 2025-01-30.
- [Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 286–292, 1978.
- [BMH⁺24] Weichen Bi, Yun Ma, Yudong Han, Yifan Chen, Deyu Tian, and Jiaqi Du. Fusionrender: Harnessing webgpu’s power for enhanced graphics performance on web browsers. In *Proceedings of the ACM Web Conference 2024*, WWW '24, page 2890–2901, New York, NY, USA, 2024. Association for Computing Machinery.
- [Bur12] Brent Burley. Physically-based shading at disney. In *ACM SIGGRAPH 2012 Courses*, 2012. Accessed: 2025-01-30.
- [Cai18] Michele Caini. Entt: Gaming meets modern c++, 2018. Accessed: 2025-01-30.
- [CBH⁺24] Satyadhyan Chickerur, Sankalp Balannavar, Pranali Hongekar, Aditi Prerna, and Soumya Jituri. WebGL vs. webgpu: A performance analysis for web 3.0. *Procedia Computer Science*, 233:919–928, 2024. 5th International Conference on Innovative Data Communication Technologies and Application (ICIDCA 2024).
- [Chr25a] Chrome for Developers. What’s new in WebGPU (Chrome 137), 2025. Accessed: February 2026.
- [Chr25b] Chrome for Developers. What’s new in WebGPU (Chrome 138), 2025. Accessed: February 2026.

Bibliography

- [Chr25c] Chrome for Developers. What’s new in WebGPU (Chrome 141), 2025. Accessed: February 2026.
- [CT82] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, 1982.
- [DPH16] Abdul Dakkak, Carl Pearson, and Wen-Mei Hwu. Webgpu: A scalable online development platform for gpu programming courses. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 942–949, 2016.
- [DPU⁺22] Landon Dyken, Pravin Poudel, Will Usher, Steve Petruzza, Jake Y. Chen, and Sidharth Kumar. GraphWaGu: GPU Powered Large Scale Graph Layout Computation and Rendering for the Web. In Roxana Bujack, Julien Tierny, and Filip Sadlo, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2022.
- [dV14] Joey de Vries. Learn opengl, 2014. Online tutorial and reference for modern OpenGL. Accessed: 2025-01-30.
- [Epi23] Epic Games. Unreal engine rendering hardware interface documentation, 2023. Unreal Engine API documentation. Accessed: 2025-01-30.
- [Fil20] Filament Team. Filament: Physically-based renderer with webgpu, 2020. Accessed: 2025-01-30.
- [Gal21] Alain Galvan. A comparison of modern graphics apis, 2021. Accessed: 2025-01-30.
- [gfx26] gfx-rs contributors. wgpu-native: Native WebGPU implementation based on wgpu-core, 2026. Accessed: February 2026.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [God23] Godot Engine Team. Godot vulkan backend, 2023. Accessed: 2025-01-30.
- [Goo26] Google. Dawn: Native WebGPU implementation, 2026. Accessed: February 2026.
- [Gra18] Niklas Gray. Handles are the better pointers, 2018. Our Machinery blog. Company acquired by Unity 2021, blog archived. Accessed: 2025-01-30.
- [Gre18] Jason Gregory. *Game Engine Architecture*. CRC Press, 3rd edition, 2018.
- [Hla24a] Helmut Hlavacs. Vienna physics engine, 2024. Physics engine companion to Vienna Vulkan Engine. Accessed: March 2026.

Bibliography

- [Hla24b] Helmut Hlavacs. Vienna vulkan engine, 2024. Educational rendering engine for University of Vienna graphics courses. Accessed: March 2026.
- [Hum24] Mark Humphrys. Ancient brain: A javascript coding platform for education with 3d graphics, websockets, ai and support for teachers. In *Proceedings of the 2024 8th International Conference on Digital Technology in Education (ICDTE)*, ICDTE '24, page 28–38, New York, NY, USA, 2024. Association for Computing Machinery.
- [Hum25] Mark Humphrys. Bringing ai apis into the classroom with a javascript coding site. *International Journal of Information and Education Technology*, 15:272–279, 01 2025.
- [HZ20] Dimitrij (Mitja) Hmeljak and Holly Zhang. Developing a computer graphics course with a game development engine. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, IT-iCSE '20, page 75–81, New York, NY, USA, 2020. Association for Computing Machinery.
- [JWB⁺25] Faris Jiwad, Owen Wolff, Omar Barabandi, Laith Najjab, and Xiaodong Qu. Flockjs: A browser-native game engine integrating webgpu and peer-to-peer networking for scalable multiplayer experiences. In *HCI International 2025*, 2025.
- [Kal24] Michael Kaltschmid. Webgpu evaluation: a comprehensive analysis of webgpu as a compute platform, 2024. ULB : Dok, University of Innsbruck institutional repository. Metadata not publicly accessible. Accessed: 2025-01-30.
- [Kar13] Brian Karis. Real shading in unreal engine 4. In *ACM SIGGRAPH 2013 Courses*, 2013. Accessed: 2025-01-30.
- [Kar23] Branimir Karadzic. bgfx: Cross-platform rendering library, 2023. Accessed: 2025-01-30.
- [Ken22] Benjamin Kenwright. Introduction to the webgpu api. In *ACM SIGGRAPH 2022 Courses*, SIGGRAPH '22, New York, NY, USA, 2022. Association for Computing Machinery.
- [Khr16] Khronos Group. Vulkan specification, 2016. Accessed: 2025-01-30.
- [KWW23] Gerald Kimmersdorfer, Dominik Wolf, and Manuela Waldner. WebGPU for Scalable Client-Side Aggregate Visualization. In Christina Gillmann, Michael Krone, and Simone Lenti, editors, *EuroVis 2023 - Posters*. The Eurographics Association, 2023.
- [LHK⁺04] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation

Bibliography

- on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, page 33–es, New York, NY, USA, 2004. Association for Computing Machinery.
- [Mic] Élie Michel. WebGPU-Cpp: A single-file zero-overhead C++ idiomatic wrapper for WebGPU native. <https://github.com/eliemichel/WebGPU-Cpp>. [Accessed 15-03-2026].
- [Mic15] Microsoft. DirectX 12 graphics overview, 2015. Accessed: 2025-01-30.
- [Mic22] Élie Michel. LearnWebGPU: WebGPU C++ guide, 2022. Accessed: 2025-01-30.
- [OHL⁺08] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [Ove23] Alexander Overvoorde. Vulkan tutorial, 2023. Accessed: February 2026.
- [PAC97] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 303–306, 1997.
- [Pal24] Saga Palmér. Performance comparison of webgpu and webgl for 2d particle systems on the web: An analysis of gpu time in web-based graphics apis. Master's thesis, KTH School of Electrical Engineering and Computer Science, Stockholm, Sweden, 2024.
- [PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.
- [Pla] PlayCanvas. Playcanvas engine with webgpu support. Accessed: 2025-01-30.
- [RGM⁺25] Alejandro Rodríguez, Pablo Luis García, Juan Sebastián Marquerie, Ricardo Marques, and Josep Blat. A cross-platform, webgpu-based 3d engine for real-time rendering and xr applications. In *Proceedings of the 30th International Conference on 3D Web Technology*, Web3D '25, page 1–9, New York, NY, USA, 2025. Association for Computing Machinery.
- [RH94] John Rohlfs and James Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 381–394, 1994.
- [RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering anti-aliased shadows with depth maps. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 283–291, 1987.

Bibliography

- [Sch94] Christophe Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.
- [Swe88] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285, 1988.
- [SWV⁺25] Sthitadhi Sengupta, Nan Wu, Matteo Varvello, Krish Jana, Songqing Chen, and Bo Han. From WebGL to WebGL: A reality check of browser-based GPU acceleration. In *Proceedings of the 2025 ACM Internet Measurement Conference*, IMC '25, page 1018–1024, New York, NY, USA, 2025. Association for Computing Machinery.
- [Thr] Three.js Team. Three.js WebGL integration. Accessed: 2025-01-30.
- [UKW23] Johannes Unterwiesingberger, Bernhard Kerbl, and Michael Wimmer. Vulkan all the way: Transitioning to a modern low-level graphics API in academia. *Computers & Graphics*, 111:155–165, 2023.
- [Uni19] Unity Technologies. Data-oriented technology stack (DOTS), 2019. First public preview released 2019. Accessed: 2025-01-30.
- [Uni22] Unity Technologies. Unity scriptable render pipeline, 2022. Accessed: 2025-01-30.
- [Ust24] Selim Usta. WebGL for 3D WebGL visualization. In *ISPRS Archives*, 2024.
- [W3C23a] W3C GPU for the Web Working Group. WebGL shading language (WGSL) specification, 2023. Accessed: 2025-01-30.
- [W3C23b] W3C GPU for the Web Working Group. WebGL specification, 2023. Accessed: 2025-01-30.
- [Web23] WebGPU Community. WebGL samples, 2023. Accessed: 2025-01-30.
- [Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, pages 270–274, 1978.
- [WMLT07] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, pages 195–206. Eurographics Association, 2007.
- [Yin] Terry Yin. Lizard: A simple code complexity analyser. <https://github.com/terryyin/lizard>. Accessed: April 2026.
- [ZSXL06] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications*, VRCIA '06, pages 311–318, 2006.

Bibliography

A. Appendix

A.1. Reproducibility Artifacts for Evaluation

This section documents the files used to reproduce [Chapter 5](#) results. All generated analysis artifacts are stored in the `data/` directory and can be regenerated from the participant-level source dataset `data/long_dataset.csv`.

The canonical repository location for this evaluation package is

<https://github.com/hlavacs/Vienna-WebGPU-Engine/tree/main/doc/evaluation>.

Direct artifact files are located under

<https://github.com/hlavacs/Vienna-WebGPU-Engine/tree/main/doc/evaluation/data>. The canonical project repository for the full engine implementation is

<https://github.com/hlavacs/Vienna-WebGPU-Engine>.

A. Appendix

Artifact	Purpose in Chapter 5
data/long_dataset.csv	Participant-level source dataset used across all evaluation analyses.
data/paired_dataset.csv	Paired pre/post records used for inferential tests and paired visualizations.
data/topic_pre_post_long.csv	Topic-wise long-format pre/post dataset for grouped topic analysis.
data/question_metadata.csv	Question/topic mapping used to aggregate questionnaire blocks consistently.
data/derived_metrics.csv	Aggregated metrics (topic means, completion counts, and threshold-based summaries).
data/h1_loc_api_comparison.csv	RQ2/H1 complexity table values (LOC and distinct API calls).
data/h1_api_calls_webgpu.txt	Distinct WebGPU API call inventory used for H1 counting.
data/h1_api_calls_vulkan.txt	Distinct Vulkan API call inventory used for H1 counting.
data/hypothesis_results.csv	Consolidated hypothesis test output (test statistics, effect sizes, corrected p-values).
data/reliability_blocks.csv	Cronbach’s α values for the Q01 to Q25 and Q26 to Q33 blocks.
data/participant_gain_ranking.csv	Participant ranking values for the H2 gain plot.
data/topic_boxplot_stats.csv	Boxplot statistics used in the H2 topic distribution figure.
data/study_level_counts.csv	Participant counts by study level used for subgroup context reporting.
data/study_group_counts_masterphd_vs_others.csv	Aggregated subgroup counts for exploratory comparisons.
data/h3_likert_distribution_q26_q33.csv	Likert distribution values for Q26 to Q33 item-level agreement reporting.
data/h3_likert_stacked_wide.csv	Stacked Likert percentages for Q26 to Q33 visualization.
data/h2_study_level_summary.csv	Exploratory subgroup summary for H2 (Master/PhD vs Others).
data/h2_study_level_test.csv	Exploratory Mann-Whitney test output for H2 subgroup comparison.
data/h3_study_level_summary.csv	Exploratory subgroup agreement rates for H3.
data/h3_study_level_test.csv	Exploratory two-proportion test output for H3 subgroup comparison.

Table A.1.: Evaluation data artifacts for reproducibility (see repository links above)

A. Appendix

The H1 LOC/API counting workflow is implemented in `scripts/h1_loc_api_comparison.py`. The script reads canonical comparison units from the LearnWebGPU-Code repository

(<https://github.com/eliemichel/LearnWebGPU-Code/tree/step030-vanilla>)

and the Vulkan Tutorial repository

(https://github.com/Overv/VulkanTutorial/blob/master/code/15_hello_triangle.cpp).

Local snapshots are stored under `temp_external/LearnWebGPU-Code/` and `temp_external/VulkanTutorial/`. The script computes the metrics and writes the three H1 output files listed in Table A.1. For WebGPU, the analysis unit is `main.cpp` plus required `requestAdapterSync/requestDeviceSync` helper implementations. Optional inspection helpers are excluded.

```
1 from pathlib import Path
2 import re
3 import pandas as pd
4
5
6 ROOT = Path(__file__).resolve().parents[1]
7 DATA_DIR = ROOT / "data"
8
9 # Canonical source units used for H1.
10 VULKAN_FILE = ROOT / "temp_external" / "VulkanTutorial" / "code" / "15_hello_triangle.cpp"
11 WEBGPU_FILE = ROOT / "temp_external" / "LearnWebGPU-Code" / "main.cpp"
12 WEBGPU_UTILS_FILE = ROOT / "temp_external" / "LearnWebGPU-Code" / "webgpu-utils.cpp"
13 WEBGPU_BRANCH = "step030-vanilla"
14
15
16 def is_code_loc(line: str) -> bool:
17     s = line.strip()
18     return bool(s) and not s.startswith("//")
19
20
21 def extract_function_body(text: str, function_name: str) -> str:
22     # Match a function signature and grab the balanced-brace body.
23     signature = re.search(rf"\b{re.escape(function_name)}\s*\([^\)]*\)\s*\{", text)
24     if not signature:
25         raise ValueError(f"Could not find function '{function_name}' in helper file")
26
27     body_start = signature.end() - 1 # Points at the opening '{'
28     depth = 0
29     for idx in range(body_start, len(text)):
30         ch = text[idx]
31         if ch == "{":
32             depth += 1
33         elif ch == "}":
34             depth -= 1
```

A. Appendix

```
35         if depth == 0:
36             return text[signature.start(): idx + 1]
37
38         raise ValueError(f"Unbalanced braces while parsing function '{
39 function_name}'")
40
41 def loc_and_calls(source_text: str, api_prefix: str) -> tuple[int, list[str
42 ]]:
43     lines = source_text.splitlines()
44     loc = sum(1 for l in lines if is_code_loc(l))
45     calls = sorted(set(re.findall(rf"\b({re.escape(api_prefix)}[A-Za-z0-9_
46 ]+)\s*\(", source_text)))
47     return loc, calls
48
49 def main() -> None:
50     DATA_DIR.mkdir(parents=True, exist_ok=True)
51
52     vk_text = VULKAN_FILE.read_text(encoding="utf-8", errors="ignore")
53     vk_loc, vk_calls = loc_and_calls(vk_text, "vk")
54
55     wgpu_main_text = WEBGPU_FILE.read_text(encoding="utf-8", errors="ignore
56 ")
57     wgpu_utils_text = WEBGPU_UTILS_FILE.read_text(encoding="utf-8", errors=
58 "ignore")
59     required_helper_text = "\n\n".join(
60         [
61             extract_function_body(wgpu_utils_text, "requestAdapterSync"),
62             extract_function_body(wgpu_utils_text, "requestDeviceSync"),
63         ]
64     )
65     # Runtime-core unit: tutorial main.cpp plus required sync request
66     # helpers.
67     # Optional inspection/reporting helpers are intentionally excluded.
68     wgpu_runtime_core = f"{wgpu_main_text}\n\n{required_helper_text}"
69     wgpu_loc, wgpu_calls = loc_and_calls(wgpu_runtime_core, "wgpu")
70
71     comparison = pd.DataFrame(
72         [
73             {
74                 "implementation": (
75                     f"WebGPU (LearnWebGPU-Code {WEBGPU_BRANCH} "
76                     "main.cpp + requestSync helpers)"
77                 ),
78                 "source_path": (
79                     "LearnWebGPU-Code/main.cpp + "
80                     "LearnWebGPU-Code/webgpu-utils.cpp"
81                 ),
82                 "non_empty_non_comment_loc": wgpu_loc,
83                 "distinct_api_calls": len(wgpu_calls),
84             },
85         ],
86     )
```

A. Appendix

```
82         "implementation": "Vulkan (VulkanTutorial 15_hello_triangle
      .cpp)",
83         "source_path": "VulkanTutorial/code/15_hello_triangle.cpp",
84         "non_empty_non_comment_loc": vk_loc,
85         "distinct_api_calls": len(vk_calls),
86     },
87 ]
88 )
89
90 comparison.to_csv(DATA_DIR / "h1_loc_api_comparison.csv", index=False)
91 (DATA_DIR / "h1_api_calls_webgpu.txt").write_text("\n".join(wgpu_calls)
92 , encoding="utf-8")
93 (DATA_DIR / "h1_api_calls_vulkan.txt").write_text("\n".join(vk_calls),
94 encoding="utf-8")
95
96 print(comparison.to_string(index=False))
97
98 if __name__ == "__main__":
99     main()
```

Listing A.1: H1 LOC and API-call counting script

A.2. Responsibility Delegation for Progressive Learning

This section documents how the Vienna engine supports top-down learning by delegating recurring technical responsibilities from learner-facing call sites into reusable internals. The comparison uses two recurring tasks (surface initialization and geometry/model loading) to show where responsibilities are placed during early tutorial steps and where detailed mechanisms can be studied later for extension. Counts are non-empty, non-comment lines within exact function boundaries. This appendix comparison is independent from H1 in [Chapter 5](#), which compares canonical WebGPU-versus-Vulkan baseline implementations. **Surface initialization responsibility.**

- LearnWebGPU raw tutorial: 21 LOC in `Application::initSurface()` (step110-next, `Application.cpp`).
- Vienna learner-facing call site: 1 LOC for `engine.initialize(options)` or 6 LOC including option assignments (`examples/tutorial/main.cpp`).
- Vienna internal abstraction: 48 LOC in `WebGPUContext::initSurface(...)` (`src/engine/rendering/webgpu/WebGPUContext.cpp`).

Geometry/model loading responsibility.

- LearnWebGPU raw tutorial: 14 LOC in `Application::initGeometry()` (step110-next, `App.cpp`).
- Vienna learner-facing call site: 1 LOC for `createModel(...)` or 4 LOC including error-check pattern (`examples/tutorial/main.cpp`).

A. Appendix

- Vienna internal abstraction: 37 LOC in `ModelManager::createModel(const std::filesystem::path&, ...)` (`src/engine/resources/ModelManager.cpp`).

This comparison should be interpreted as delegation of responsibility rather than elimination of complexity. In the Vienna engine, complexity is intentionally shifted from user code into reusable engine internals, where platform branching and resource orchestration are implemented once and then reused by all examples.

A.3. AI Tool Usage Documentation

This appendix provides detailed documentation of artificial intelligence tools used during the creation of this thesis, as required by academic integrity standards. The following table summarizes the tools employed, their purposes, affected chapters, and documentation references.

Tool	Purpose	Location	Documentation
GitHub Copilot Chat (Claude Sonnet 4.5)	Linguistic refinement, sentence structure	All Chapters	Custom Prompts
GitHub Copilot Chat (Claude Sonnet 4.5)	Bibliography management, citation formatting, reference discovery	2	<code>bibliography.bib</code>
GitHub Copilot Chat (Claude Sonnet 4.5)	LaTeX formatting compliance (e.g., figures), template adherence	All Chapters	Source <code>.tex</code> files
Grammarly	Linguistic refinement	All Chapters	Source <code>.tex</code> files
GitHub Copilot Chat (Claude Sonnet 4.5)	Code documentation, comment generation, API reference lookup	Engine Implementation	Inline code comments, documentation strings
Chat GPT-5	Understanding GPU concepts, finding best practices, implementation patterns	Engine Implementation	Design document references, architecture notes

Table A.2.: Overview of AI tools used during thesis creation

Nature of AI Assistance

The AI assistance provided by GitHub Copilot Chat was limited to the following categories:

A. Appendix

Writing and Structuring: The AI tool assisted in formulating complex technical descriptions, suggesting sentence structures for clarity, organizing sections logically within chapters, and ensuring consistent academic writing style throughout the document. All technical content, research findings, and conclusions originate from the author’s work.

Literature Review Support: The tool helped synthesize information from multiple sources into coherent paragraphs, suggested relevant papers based on topic descriptions, and assisted in formatting citations according to BibTeX standards. All source material was independently reviewed and evaluated by the author for relevance and quality.

Code Documentation: During engine development, the tool generated documentation comments for functions and classes based on implementation details, suggested descriptive variable and function names following C++ conventions, and helped format code examples for inclusion in the thesis. All implementation logic, algorithms, and architectural decisions were designed and coded by the author.

Concept Understanding: The tool provided explanations of complex GPU programming concepts such as bind group layouts, pipeline states, and synchronization primitives, helped locate relevant sections in WebGPU specification documents, and suggested best practices from established graphics programming resources. All concepts were independently studied and verified through documentation and experimentation.

LaTeX Formatting: The tool ensured compliance with thesis template requirements, prevented formatting errors such as improper line breaks, and maintained consistent style throughout the document.

Author’s Original Contributions

The following aspects of this thesis represent the author’s independent intellectual and creative work, where AI tools were used only for feedback, repetitive tasks, and code documentation as described above, including but not limited to:

- Complete design and implementation of the Vienna WebGPU Engine, including all architecture decisions, C++ source code (over 10,000 lines of code not including comments), WGSL shaders, and build system configuration
- Development of the factory-based resource management architecture and type-safe handle system
- Implementation of the Unity-inspired scene graph system with hierarchical transforms and lifecycle management
- Creation of the physically-based rendering pipeline including Cook-Torrance BRDF, shadow mapping for multiple light types, normal mapping, and post-processing effects
- Design and implementation of all engine subsystems including texture loading, mesh processing, material management, input handling, and debug visualization

A. Appendix

- Integration of cross-platform support for native Windows and web deployment via Emscripten
- Formulation of research questions, hypotheses, and contributions
- Design of evaluation methodology including user study protocol, questionnaire development, and performance benchmarking framework
- Independent selection, review, and critical evaluation of all academic and technical sources in the literature review
- Analysis and interpretation of all experimental results, performance data, and user feedback
- All technical decisions regarding API selection, architectural patterns, implementation strategies, and optimization approaches
- Development of comprehensive code documentation, architecture diagrams, and example applications with educational intent

Verification

I declare that I have used AI tools only as described above and that all intellectual contributions, creative work, research design, and technical implementation are my own. The AI tools served solely as assistive writing and documentation aids and did not generate research findings, experimental results, or technical solutions.

Vienna, 29th June 2026

Kalian Danzer