COMP7705 Project
Detailed Project Proposal

Project Title:	Game Development Framework with Vulkan Rendering
Mentor:	Dr. Loretta Y.K. Choi
Student 1 (Leader)	Zeng Zhongze
Student 2	Lu Jiaxiang
Student 3	Yang Ruitian
Student 4	Tian Qijia
Student 5	

Aim

This project aims to provide a comprehensive and accessible foundation for developers, combining advanced rendering techniques, flexible scripting capabilities, and a user-friendly framework to accelerate the creation of high-quality 3D games and applications.

Key Objectives:

1. **Advanced Rendering Techniques:** Implement state-of-the-art rendering methods from recent CG papers.
2. **Scripting Language Integration:** Provide a scripting language interface (C# or JavaScript) that allows users to create custom game logic, initialize models, handle game loops, and manage events within the framework.
3. **User-Centric Framework:** Design a flexible framework that enables users to build their own game logic on top of it, facilitating the creation of diverse and interactive experiences.
4. **Working Example:** Deliver a working example that demonstrates the framework's capabilities by building a simple yet engaging interactive game, showcasing its potential for real-world applications.
5. **Future Enhancements:** Plan for future integrations, such as water physics, dynamic lighting, and plant rendering, to further enrich the framework's capabilities and provide a powerful foundation for developers to build immersive and interactive 3D experiences.

Brief Literature Review

Modern game engine architectures have undergone a significant transformation over the past decade, shifting from traditional object-oriented paradigms to more data-driven designs. This shift is evidenced by the adoption of Entity Component Systems (ECS) in modern engines such as Bevy, which emphasizes performance-focused data organization and modularity. Researchers have argued that decoupling data from behavior can lead to significant performance gains by leveraging parallel processing and improved cache coherence in modern hardware, though the extent of these benefits varies with specific implementation and use case.

Bevy implements its ECS with simple Rust structs for components and Rust functions for systems, utilizing a distinctive "Schedule-First" design paradigm that facilitates controlled parallel execution. While this approach enables efficient cache-friendly operations, the parallelism is constrained by cross-system data dependencies, which can limit performance in certain scenarios. Although benchmarks suggest Bevy's ECS performs well, its reliance on Rust's ownership model introduces a steep learning curve for developers accustomed to higher-level languages. Additionally, the absence of a built-in editor poses challenges for interdisciplinary teamwork, though community-developed tools like Bevy Editor and Bevy Inspector Egui have emerged to address these limitations.

Commercial engines have evolved significantly in response to data-oriented design trends. Unity has developed its Data-Oriented Technology Stack (DOTS) with an ECS implementation, Job System, and Burst Compiler, moving beyond its traditional component-based architecture while maintaining C# as its scripting language. Despite these advances, Unity's closed-source nature and licensing constraints continue to limit deep customization. Unreal Engine, while known for high-end PC graphics, has made substantial progress in mobile optimization with its Nanite and Lumen technologies now adapted for mobile platforms. Its recent introduction of the Mass Entity framework also demonstrates a shift toward ECS principles, while maintaining the accessibility of its Blueprint visual scripting system.

Graphics APIs and rendering techniques play a vital role in modern engine performance, with selection highly dependent on project requirements. Vulkan provides lower-level hardware access with explicit multi-threading support, significantly reducing CPU overhead compared to legacy APIs like OpenGL. However, in smaller projects or those with simple rendering requirements, OpenGL may offer sufficient performance with lower implementation complexity. Recent advances in rendering, such as real-time volumetric cloud rendering that combines single and multiple scattering techniques, demonstrate the potential for achieving both high realism and efficiency in complex scenes, though these techniques often require careful optimization for specific hardware targets.

Another critical aspect in engine design is the incorporation of scripting languages to enhance developer productivity. Separating performance-critical systems from game logic allows developers to utilize higher-level languages like C#, JavaScript, or Lua. In particular, LuaJIT can achieve near-native performance in specific optimization scenarios. This hybrid approach has been adopted by both commercial and open-source engines, with notable improvements in the latter category. Godot 4, for example, has significantly enhanced its rendering capabilities while maintaining accessibility through GDScript and offering C# support, narrowing the gap with commercial alternatives through its GDExtension system that efficiently bridges high-level scripting with native code.

Proposed Methodology

The development of our game engine will follow a structured approach combining modern systems design with performance-focused architecture principles. Our methodology emphasizes data-oriented design, modular composition, and platform abstraction while maintaining developer accessibility.

Core Architecture Development

We will implement an Entity Component System (ECS) architecture inspired by Bevy's design principles but adapted for our C++ environment. This data-oriented approach will organize game objects as entities composed of data-only components, with behavior implemented through separate system functions. This architecture facilitates cache-coherent memory access patterns and enables parallel processing on modern multi-core hardware. The implementation will include a custom archetype-based storage system that groups entities with identical component types to maximize memory locality.

For modular extensibility, we will develop a plugin-based architecture allowing engine subsystems to be implemented as self-contained modules with well-defined interfaces. This approach enables components to be added, removed, or replaced without affecting the core system, supporting a wide range of use cases from lightweight mobile applications to graphically intensive desktop games. The plugin system will manage dependencies between modules and handle initialization order automatically.

The rendering pipeline will be implemented using a flexible render graph system that abstracts graphics API details while optimizing specifically for Vulkan when available. This graph-based design enables developers to define complex rendering workflows as directed acyclic graphs (DAGs) of render passes, allowing for automatic resource barrier management and execution optimization. The render graph will dynamically compile to efficient command buffers, maximizing GPU utilization through techniques such as asynchronous compute and multi-queue submission.

Key Component Implementation

The core engine layer will feature a custom memory management system optimized for game development patterns, including allocators specialized for specific usage scenarios: frame allocators for transient per-frame data, pool allocators for frequently allocated game objects, and a general-purpose allocator for persistent engine data. An event system with both immediate and queued dispatch modes will enable decoupled communication between engine components. Thread management will utilize a task-based parallelism model where work is divided into small tasks that can be executed concurrently by a thread pool, with sophisticated dependency tracking to minimize synchronization points.

Our rendering system will be built on a Vulkan abstraction layer that provides a unified API while exposing Vulkan-specific features when needed. For platforms without Vulkan support, we'll implement fallback paths using MoltenVK(iOS/macOS). The system will incorporate modern rendering techniques from recent computer graphics research, particularly focusing on physically-based rendering, real-time global illumination approximations, and volumetric effects. Material and shader management will utilize a node-based approach that allows artists to create complex materials visually while generating optimized shader code behind the scenes.

For scripting integration, we will implement C# bindings (using Mono or .NET Core) to provide robust object-oriented programming capabilities. The script API design will offer a clean interface to engine components while abstracting low-level details, and hot-reloading support will enable rapid iteration during development. Script performance will be enhanced through ahead-of-time compilation options and native code generation for performance-critical sections. This approach balances development efficiency with runtime performance, allowing developers to express game logic in a high-level language while maintaining efficient interaction with underlying engine systems.

The asset pipeline will feature a format-agnostic loading system that accepts industry-standard formats while converting them to optimized runtime representations. This system will support automatic LOD generation, texture compression, and mesh optimization. Resource management will include reference counting, asynchronous loading, and streaming capabilities for large open worlds. Hot-reloading will be implemented comprehensively across the engine, allowing developers to modify assets, code, and configuration at runtime.

Testing and Validation

Our methodology includes continuous testing throughout development. We will employ automated unit tests for core engine components, performance benchmarks to identify bottlenecks, and integration tests for cross-module functionality. Real-world validation will be conducted through the development of sample applications targeting different use cases and platforms, with feedback incorporated into the engine design iteratively.

By following this methodology, we aim to create a game engine that combines the performance benefits of data-oriented design with the accessibility of modern development workflows, providing a solid foundation for a wide range of interactive applications.

Milestones

Tasks		*Estimated completion time*	*Estimated number of learning hours*
1	**Core Framework Design & Infrastructure** Design ECS core architecture Implement basic memory management system Build plugin system framework Set up project environment and build system	Week 1-2	30
2	**ECS Implementation & Thread Management** Implement entity, component, and system base classes Develop scheduling system (Schedule-First design) Build basic thread pool and task allocation mechanism Complete event system design	Week 3-4	35
3	**Rendering System Foundation** Develop Vulkan abstraction layer Implement basic rendering pipeline Create simple render graph system Design material and shader management framework	Week 5-6	40
4	**Resource Management System** Implement resource loading and caching system Develop hot-reloading mechanism Build format-agnostic asset manager Create basic scene serialization system	Week 7-8	35
5	**Scripting Engine Integration** Implement C# bindings (using Mono or .NET Core) Design scripting engine API interface Establish communication between scripts and ECS	Week 9-10	40

6	**Advanced Rendering Features** Implement physically-based rendering system Develop lighting and shadow systems Integrate atmospheric scattering techniques Implement post-processing pipeline	Week 11-12	40
7	**Debugging & Development Tools** Build performance profiling tools Implement visual debugging interface Develop component inspector Create basic scene editing functionality	Week 13-14	30
8	**Optimization & Cross-Platform Support** Implement rendering optimization techniques Add graphics API fallback support (MoltenVK) Optimize memory and performance Establish cross-platform build pipeline	Week 15-16	20
9	**User Testing & Feedback Collection** Conduct usability tests with target users. Collect feedback for iterative improvements.	Week 17-18	15
10	**Final Documentation & Demo** Prepare user manuals and technical reports. Record a system demonstration video.	Week 19-20	15
			Total: 300

Deliverables

Items	
1	**User Manual** – Step by step guide for installation and usage.
2	**Technical Report** – Detailed methodology, test results, and system architecture.
3	**Demo Video** - Showcasing system features and use cases.
4	**Final Presentation Slides** - Summary of project goals, innovations, and outcomes.
5	**Basic Framework –** Basic game engine on top of Vulkan API
6	**Game Demo -** Game program developed based on the framework above.
7	
8	
9	
10	