Shrike : Framework for Game Development and Real time Simulations

Submitted in partial fulfillment of the requirements

for the degree of

B.E. Computer Engineering

By

Evan Mendonsa	**20**	**212060**
Nigel Colaço	**24**	**212013**
Jason Sampy	**34**	**212098**
Aakhaash Diclas	**40**	**212023**

Guide

Prof. Nidhi Gaur

Assistant Professor

St. Francis Institute of Technology

(Engineering College)

An Autonomous Institute, Affiliated to University of Mumbai

NAAC A+ Accredited | CMPN, EXTC, INFT NBA Accredited | ISO 9001:2015 Certified

Department of Computer Engineering

(AY 2024-2025)

SFIT LIBRARY 2025

CERTIFICATE

This is to certify that the project entitled **"Shrike : Framework for Game Development and Real time Simulations"** is a bonafide work of **"Evan Mendonsa(20), Nigel Colaço(24), Jason Sampy(34) and Aakhaash Diclas(40)"** submitted to the University of Mumbai in partial fulfillment of the requirement for the award of the degree of B.E. in Computer Engineering.

Prof. Nidhi Gaur
Guide

Dr. Kavita Sonawane
Head of Department

Dr. Sincy George
Principal

SFIT LIBRARY 2025

Project Approval Report for B.E.

This project report entitled **"Shrike : Framework for Game Development and Real time Simulations"** by **Evan Mendonsa(20), Nigel Colaço(24), Jason Sampy(34) and Aakhaash Diclas(40)** is approved for the degree of **B.E. in Computer Engineering.**

Examiners

1. _____

2. _____

Date: 30/04/25

Place: Mumbai

SFIT LIBRARY 2025

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Evan Mendonsa (20)

Nigel Colaço (24)

Jason Sampy (34)

Aakhaash Diclas (40)

Date: 30/04/25

SFIT LIBRARY 2025

Abstract

Shrike is a modern game development framework designed to simplify the creation of web-based games, simulations, and 3D visualizations. It integrates a real-time physics engine, advanced material simulation, dynamic rendering pipeline, and a lightweight scripting system, enabling developers to build complex projects directly within a browser environment. The framework offers a high level of abstraction through a modular Entity-Component-System (ECS) architecture, allowing users to focus on creativity rather than low-level technical details. Shrike's browser-based scene editor streamlines workflows by providing real-time object manipulation, material editing, physics property management, and live scene previews without the need for external tools. Built with WebGL2 and JavaScript, Shrike ensures broad platform compatibility, fast loading times, and efficient GPU utilization. The framework is ideal for indie developers, web designers, and 3D modelers, supporting physically based rendering (PBR), real-time physics simulation, and customizable scripting functionalities. By bridging the gap between technical complexity and creative freedom, Shrike empowers users to rapidly prototype, develop, and deploy interactive 3D applications, offering a lightweight, accessible, and powerful solution for the next generation of web-based interactive experiences.

Keywords: Shrike, Game framework, Web-based 3D applications, 3D Simulations, Physics engine, Material simulation, Real-time rendering, Scripting System, ECS, Browser-based development

SFIT LIBRARY 2025

Contents

Declaration iii

Abstract iv

1 Introduction 1

1.1 Introduction . . . 1

1.2 Background Study Terminologies/Definitions of New Terms . . . 1

1.3 Fundamental Study Points of the Selected Topic and the Domain . . . 2

1.4 Identification of Challenges in the Selected Topic . . . 2

1.5 Problem Statement and Proposed Solution . . . 3

1.6 Scope of the System . . . 3

2 Literature Survey 4

2.1 Survey of Existing Systems . . . 4

2.2 Limitations of Existing Systems or Research Gaps . . . 5

2.3 Motivation . . . 6

3 Proposed System: Analysis 7

3.1 Detailed Explanation of Proposed System . . . 7

3.1.1 Workflow Diagram . . . 7

3.1.2 Working Principle . . . 8

3.1.3 Module-wise Explanation . . . 9

3.2 System Analysis . . . 10

3.2.1 Functional Requirements . . . 10

3.2.2 Non-Functional Requirements . . . 11

3.2.3 Specific Requirements . . . 12

3.2.4 Software and Hardware Requirements . . . 12

3.2.5 Use Case Modeling . . . 13

SIT LIBRARY 2025

3.3 Proposed System: Analysis, Modeling, and Design 14
3.3.1 Activity Diagrams 14
3.3.2 DFD Diagrams 16
3.3.3 Architectural View 19
3.3.4 Algorithms/Methodology 20
3.3.5 UI/UX Design 20

4 Implementation Plan and Experimental Setup of the Proposed System 23
4.1 Experimental Setup 23
4.1.1 Details Discussion of Input/Dataset 23
4.1.2 Performance Evaluation Parameters 25
4.2 Code for Sem VIII 28

5 Results and Discussions 37
5.1 Presentation and Validation of Results 37
5.1.1 Device Specifications 38
5.1.2 Performance Metrics 39
5.2 Comparative Analysis with Existing Systems 39

6 Conclusion 41

References 42

Acknowledgments 43

SFIT LIBRARY 2025

List of Figures

3.1 Block Diagram of Proposed System/Workflow 7
3.2 Internal Architecture of Shrike 9
3.3 Use Case Diagram 13
3.4 UML Activity Diagram of Shrike Engine Framework 15
3.5 DFD Level 0 (Context Diagram) 16
3.6 DFD Level 1 (Major Subsystems) 17
3.7 DFD Level 2 (Detailed Component Interaction) 18
3.8 Architectural Diagram of Shrike Engine 19
3.9 Initial Editor View 21
3.10 Startup Editor View 22

4.1 Project Directory Structure 28
4.2 Core System (PART 1) 29
4.3 Core System (PART 2) 30
4.4 Physics System 31
4.5 Renderer (PART 1) 32
4.6 Renderer (PART 2) 33
4.7 Script System 34
4.8 Shader (PART 1) 35
4.9 Shader (PART 2) 36

5.1 Scene Creation in Shrike Editor: Developer Building a Flappy Bird Game . . . 37
5.2 Gameplay Execution in Shrike: Flappy Bird Game Running Smoothly 38

SIT LIBRARY 2025

List of Tables

3.1 Use Case 1: Create New Game Object 13
3.2 Use Case 2: Attach Script to Game Object 14
3.3 Use Case 3: Save Scene (Serialization) 14

5.1 Performance Metrics for Flappy Bird Game Execution on Shrike 39
5.2 Comparative Analysis of Game Engine vs Three.js 40

SFIT LIBRARY 2025

Chapter 1

Introduction

1.1 Introduction

Shrike is a framework developed to streamline the creation of web-based games and real-time simulations. It bridges the gap between simplicity and high-performance. Shrike is crafted to cater specifically to indie game developers, web designers, and 3D modelers by providing essential features like a physics engine, real-time rendering, and a script compiler, enabling them to focus on creative expression rather than complex technical implementation. This focus on user-friendly design and powerful capabilities makes Shrike a versatile tool for a wide range of interactive projects, from intricate game worlds to detailed scientific simulations. The framework's architecture is designed to be modular and extensible, allowing developers to customize and expand its functionality to suit their specific needs. This adaptability is crucial in the fast-evolving landscape of web technologies, ensuring that Shrike can remain relevant and effective for future projects.

1.2 Background Study Terminologies/Definitions of New Terms

With the rise of indie games—over 13,000 titles produced in recent years compared to fewer releases by major studios—there's a substantial need for accessible development tools. This surge in indie game development underscores a shift in the gaming industry, where smaller, independent teams are producing innovative and creative content. However, these teams often operate with limited resources and technical expertise compared to larger studios. They frequently rely on efficient, user-friendly tools to bring their visions to life. Adobe Flash once facilitated easy web-based game development, but its discontinuation created a void in the gaming ecosystem. Flash provided a relatively low barrier to entry for creating interactive content, including games, directly within the browser. Its absence has left many developers seeking alternative solutions. Shrike steps in to provide a robust, all-in-one solution for developing 3D browser games, filling this void with a streamlined, intuitive interface for scene editing and physics simulation. This aims to empower developers to create immersive 3D experiences without needing to master complex, disparate tools. By integrating these essential features into a single framework, Shrike simplifies the development process and allows creators to focus on the artistic and design aspects of their projects.

SIT LIBRARY 2025

1.3 Fundamental Study Points of the Selected Topic and the Domain

Shrike represents the convergence of high-performance web technologies, including:

- **3D JavaScript Engine:** Provides the core engine for managing 3D scenes. This engine handles the rendering, animation, and overall logic of the 3D environment. It is the foundation upon which the entire game or simulation is built. The engine is designed to be efficient and flexible, supporting a wide range of 3D graphics and effects. It also provides a robust API for developers to interact with the scene and control its behavior.
- **Physics Simulation:** Implements a realistic physics engine optimized for browser performance. This includes features like collision detection, gravity, and other physical interactions, adding a layer of realism and interactivity to the developed content. The optimization for browsers is crucial for ensuring smooth performance across different platforms, including desktops, laptops, and mobile devices. This ensures a consistent user experience regardless of the hardware.
- **Scene Editor:** Allows developers to build and modify game environments interactively. This visual tool simplifies the process of creating and adjusting the layout, objects, and elements within the 3D scene, making it more intuitive and efficient. The scene editor provides a WYSIWYG (What You See Is What You Get) interface, allowing developers to see the results of their changes in real-time. This significantly speeds up the development process and reduces the likelihood of errors.

1.4 Identification of Challenges in the Selected Topic

The discontinuation of Adobe Flash has created a significant gap in web-based game development, leaving indie developers without an accessible, unified platform that combines ease of use with high-performance 3D capabilities. This gap has forced developers to adopt more complex and fragmented workflows. Current solutions require developers to navigate fragmented tools and complex workflows across multiple platforms, creating technical barriers that limit their ability to fully realize their creative visions in browser-based 3D games. The need to manage different tools for different aspects of development (e.g., separate tools for 3D modeling, physics, and rendering) can significantly increase development time and complexity. This fragmentation also makes it more difficult to collaborate on projects, as developers may need to use different software and file formats. Furthermore, the lack of a standardized platform can lead to inconsistencies in performance and compatibility across different browsers and devices.

SLIT LIBRARY 2025

1.5 Problem Statement and Proposed Solution

Developing a high-performance 3D JavaScript game engine that provides indie developers with an integrated development environment, combining an intuitive scene editor, physics simulation capabilities, and optimized browser rendering to enable efficient creation of web-based 3D games without requiring expertise across multiple platforms and frameworks, will prove to be revolutionary. This solution addresses the core issue of fragmented development by providing a single, unified tool. By integrating all the necessary components into one platform, Shrike eliminates the need for developers to learn and manage multiple different software packages. This not only simplifies the development process but also reduces the learning curve for new developers, making 3D game development more accessible to a wider audience.

1.6 Scope of the System

The scope of the Shrike game framework includes:

- Providing a versatile and high-performance environment for game development. This involves offering a comprehensive set of tools and features that support the creation of a wide variety of game genres and styles. This includes support for different types of 3D models, textures, and animations, as well as tools for creating and managing game logic. Shrike aims to be flexible enough to accommodate the needs of both novice and experienced game developers.
- Enabling web-based simulation. Beyond games, Shrike is intended to facilitate the development of interactive simulations for educational, scientific, and other purposes. This could include simulations of physical systems, biological processes, or even social interactions. The ability to create web-based simulations opens up a wide range of possibilities for using interactive 3D graphics in various fields.
- Supporting real-time rendering. This ensures that the visual elements of the game or simulation are displayed smoothly and dynamically, enhancing the user experience. Real-time rendering is crucial for creating immersive and engaging experiences, whether it's a fast-paced action game or a detailed scientific visualization. Shrike is designed to optimize rendering performance, taking advantage of modern browser capabilities and hardware acceleration.
- Integrating multiple components, including a physics engine, material simulation, and a rendering system. This integration is key to providing a seamless and efficient development workflow and also ensures that all the different components of the framework work together smoothly and efficiently, reducing the likelihood of conflicts or errors.

SFIT LIBRARY 2025

Chapter 2

Literature Survey

2.1 Survey of Existing Systems

The advancement of real-time 3D graphics and browser-based applications has significantly influenced game engine development. A variety of frameworks and research prototypes have attempted to streamline this process by focusing on modular architectures, high-performance rendering, lightweight scripting, and cross-platform compatibility. The Shrike framework builds upon these innovations, filling critical gaps identified in previous systems. This section explores key systems and studies that have shaped the design of Shrike.

Liu et al. [1] proposed a GPU-accelerated WebGL model rendering engine capable of real-time image generation with programmable shaders. While the study successfully demonstrates cross-platform rendering, its focus on static scenes excludes dynamic game environments, limiting its use as a full-fledged game engine.

Jiang and Niu [2] presented an adaptive binary tree (ABT) approach for view frustum culling, reducing rendering time by 48%. Shrike integrates similar spatial partitioning strategies for optimizing large scene rendering and improving overall frame rates.

Pouhela et al. [3] introduced an Entity Component System (ECS) tailored for IoT brokers. Although not originally intended for games, their architecture offered scalability, modularity, and energy efficiency—principles that directly informed Shrike's ECS-based design, promoting reusability and decoupled logic.

Mohd et al. [4] conducted a comparative study of Unity, Unreal, Godot, and CryEngine, highlighting their extensibility, plugin support, and active ecosystems. However, these engines often demand significant hardware resources and are ill-suited for lightweight or browser-native use, a gap Shrike aims to bridge.

Chernousenko et al. [5] developed a physics engine for fluid and gas simulation that operates efficiently on mid-range systems. Although focused on fluid dynamics, the system's design principles—especially memory-efficient calculations—are reflected in Shrike's rigid-body simulation module.

Wiryadi and Kosala [6] showed that GPU-based geometry shaders can accelerate particle rendering by 2.5x compared to CPU-bound methods. This approach is central to Shrike's visual effects system for handling complex scene elements like smoke and fire.

SIT LIBRARY 2025

Rajendran [7] emphasized the importance of profiling JavaScript performance across browsers. The findings stress optimization of JIT compilation, memory management, and asynchronous calls—considerations embedded in Shrike's real-time scripting engine.

Shukla [8] explored the evolution of JavaScript as a full-stack development language. With support for ES6 modules, async features, and broad library ecosystems, JavaScript's maturation underpins Shrike's decision to use it for core engine logic.

PlayCanvas [9] is a commercial cloud-based WebGL game engine offering a collaborative IDE and modular components. However, its dependency on cloud access and limited offline capabilities contrast with Shrike's fully standalone, open-source architecture.

Babylon.js [10] demonstrates powerful physics, shader, and asset tools for the browser, but its steep learning curve and broader scope can overwhelm new developers. Shrike addresses this by maintaining minimalism while retaining core engine features.

Wang et al. [11] proposed adaptive LOD (Level of Detail) management systems for real-time 3D applications. Shrike applies similar LOD techniques to optimize GPU workload and maintain consistent frame rates for complex game scenes.

The SyDRA study [12] outlined how game engine subsystems interact and impact performance. This comprehensive system design approach influenced Shrike's modular integration of physics, rendering, and input systems for efficiency and scalability.

Godot and Unity both employ visual scripting and node-based scene graphs. Shrike adopts a simplified version of this through a WebGL scene editor that includes parent-child transformations and spatial hierarchy traversal [13].

Epic Games' documentation on physically-based rendering (PBR) [14] explains the importance of properties like roughness, metallicity, and specular reflection. Shrike incorporates a simplified PBR pipeline using WebGL2 shaders to simulate real-world material responses.

The ANGLE project [15] used in Chromium browsers abstracts hardware differences and ensures consistent WebGL behavior across platforms. Shrike's rendering engine is optimized with these behaviors in mind, ensuring robust cross-browser compatibility.

2.2 Limitations of Existing Systems or Research Gaps

Despite advancements in game engine design, notable limitations persist in existing systems:

- **Lack of Web-Native Integration:** Most engines are either desktop-focused (Unity, Unreal) or cloud-reliant (PlayCanvas), making them less ideal for offline or browser-only development workflows.
- **Fragmented Architecture:** Tools for scripting, rendering, and physics often exist in isolation, requiring developers to manually integrate third-party systems, thereby increasing complexity.

SJIT LIBRARY 2025

- **High Learning Curve:** Engines like Babylon.js and Godot offer rich toolsets but can be overwhelming for beginners or non-technical users, impeding rapid prototyping.
- **Performance Inconsistencies:** JavaScript-based game engines often suffer from unpredictable behavior across browsers, largely due to differences in rendering backends and memory handling.
- **Limited Real-Time Editing:** Most web engines lack in-browser scene editors with live preview and feedback, which hinders fast iteration and testing.
- **Shallow ECS Adoption:** Though ECS architectures have shown promise, their adaptation for dynamic, real-time games remains limited and underexplored in existing research.

2.3 Motivation

The surge of indie game development and the discontinuation of tools like Adobe Flash have created a significant void in browser-first game engines. Existing systems either impose high system requirements or require installation, plugins, or platform-specific knowledge. This limits their accessibility, especially for web developers and hobbyists.

Shrike is motivated by the need for a modular, lightweight, browser-native 3D game engine that integrates modern principles—such as ECS architecture, WebGL2 rendering, and real-time scripting—within a single development environment. The system enables real-time prototyping, simulation, and deployment without requiring users to leave the browser or write complex boilerplate code.

Shrike's embedded scene editor, support for physically-based materials, and dynamic component-based logic make it uniquely positioned for educational use, indie game development, and simulation projects that require rapid experimentation and visual fidelity.

By addressing the fragmentation and accessibility issues present in current engines, Shrike provides an elegant solution that democratizes 3D game development.

SFIT LIBRARY 2025

Chapter 3

Proposed System: Analysis

3.1 Detailed Explanation of Proposed System

3.1.1 Workflow Diagram

The proposed system, Shrike – Framework for Game Development and Real-time Simulations, follows a modular, highly interactive design approach to enable efficient game creation and simulation directly in the browser.

The workflow, as illustrated in Figure 3.1, revolves around two major participants: the User/Developer and the Engine Core. The user interacts with multiple interfaces (scene graph, resource management, property management, physics system), and through serialization and deserialization processes, manipulates game objects, assets, and properties. These changes are fed into the Engine Interface, which processes and displays the final Results.

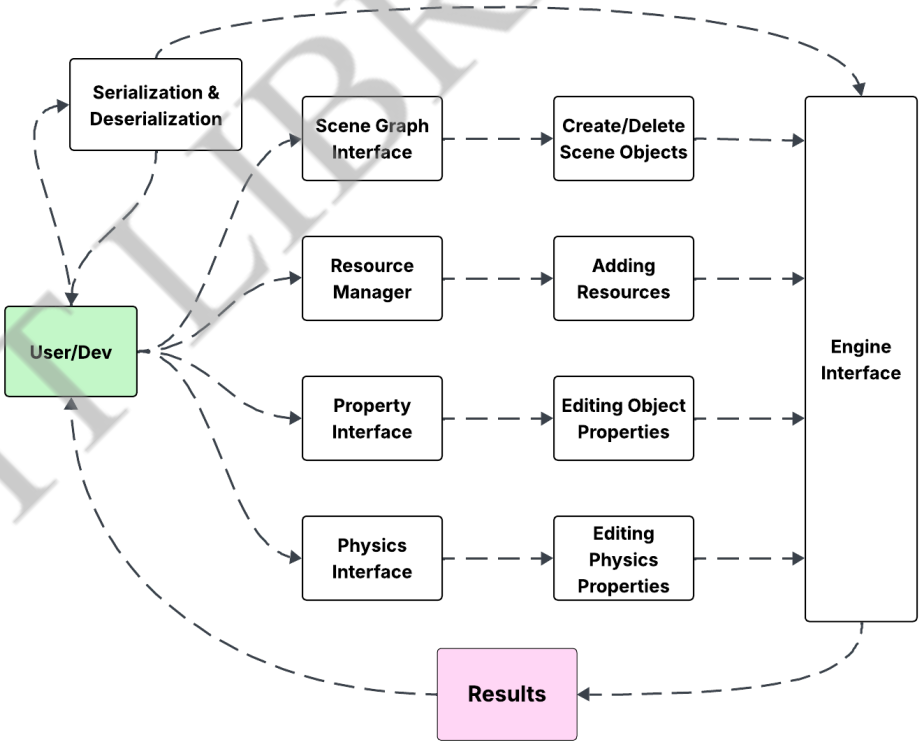


Figure 3.1: Block Diagram of Proposed System/Workflow

Key Highlights of Workflow:

SFIT LIBRARY 2025

- **Serialization & Deserialization** allows users to load or save entire scenes.
- **Scene Graph Interface** manages object hierarchies, allowing creation or deletion of scene objects.
- **Resource Manager** handles asset management (models, textures, sounds).
- **Property Interface** provides tools for editing object attributes (transformations, materials).
- **Physics Interface** allows setting physical behaviors (gravity, collision responses).
- Final execution flows into the **Engine Interface**, which computes frames and visual outputs based on the user's inputs.

This modular interaction model ensures a flexible, scalable system, capable of adapting to different development needs like prototyping, simulation, or full game deployment.

3.1.2 Working Principle

The working principle of Shrike is based on a modular real-time engine loop, integrating rendering, physics, scripting, and event handling. It follows these major steps:

1. **Initialization Phase:**
    - Load essential engine components.
    - Initialize Scene Manager, Physics Engine, Renderer, Script Engine, and Resource Manager.
    - Setup GPU resource pools and allocate memory buffers.
2. **Asset and Scene Setup:**
    - Load or create game objects via the Scene Graph Interface.
    - Attach components such as MeshRenderer, Collider, Rigidbody to entities.
    - Load textures, shaders, and materials using the Resource Manager.
3. **Main Game Loop Execution:**
    - Event Handling: Capture and process user input events (keyboard, mouse, touch).
    - Script Execution: Dynamically update entity behaviors through user-defined scripts.
    - Physics Update: Simulate object interactions using collision detection and rigidbody integration.
    - Component Updates: Synchronize entity states (transform, materials) based on script and physics outputs.

SET LIBRARY 2025

- Rendering Phase: Execute the rendering pipeline — frustum culling, LOD selection, draw call batching.
- Result Presentation: Display computed frames in real-time on the canvas.

4. **Serialization/Deserialization Phase:**
    - Enable saving/loading entire scenes, including objects and components, using JSON formats.
5. **Termination Phase:**
    - Deallocate GPU resources, clean up memory, and safely shut down the engine loop.

3.1.3 Module-wise Explanation

The internal architecture of Shrike is divided into focused modules. The overall structure is shown in the Figure 3.2 :

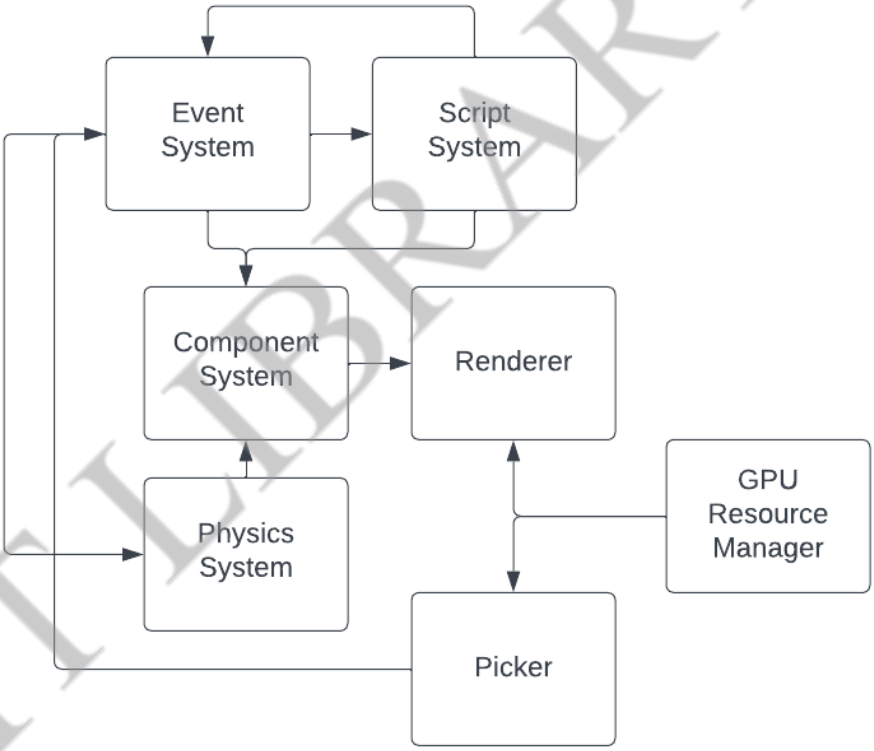


Figure 3.2: Internal Architecture of Shrike

The internal architecture of Shrike is divided into focused modules:

- **Event System:** Captures and processes user-generated events such as keyboard inputs, mouse clicks, touch gestures. These are dispatched to the relevant subsystems for handling interactions and gameplay logic.

SFIT LIBRARY 2025

- **Script System:** Executes user-defined scripts attached to game objects. Supports modular behavior attachment without modifying the engine core.
- **Component System:** Manages all game entities and their attached components (e.g., Transform, MeshRenderer, Collider, Light). Ensures synchronization between logic, physics, and rendering subsystems.
- **Physics System:** Simulates real-world physics, handling collision detection (AABB, Broadphase/Narrowphase) and rigid-body dynamics. Updates object states according to physical interactions.
- **Renderer:** Handles the entire rendering pipeline: scene traversal, shader execution, material application, post-processing effects. Optimized for WebGL2 using GPU instancing and deferred rendering where necessary.
- **Picker:** Supports object picking by mapping screen-space clicks to world-space objects, crucial for editor interactions and gameplay mechanics.
- **GPU Resource Manager:** Manages low-level GPU operations, such as buffer creation, texture management, and shader pipeline compilation, ensuring optimal performance across devices.

Each module is loosely coupled but highly cohesive, meaning they operate independently but work seamlessly together via well-defined interfaces. This ensures that developers can extend or modify functionalities (like custom physics behaviors or new shaders) without altering the core engine design.

3.2 System Analysis

3.2.1 Functional Requirements

The proposed system — Shrike — is designed to provide an end-to-end environment for browser-native 3D game development and simulations. The core functional requirements include:

- **Scene Management:**
  - Create, update, and delete game objects within the scene graph.
  - Load and save scenes via serialization/deserialization (JSON-based).
- **Resource Management:**
  - Load assets such as 3D models, textures, audio files dynamically.
  - Manage GPU resources for efficient memory usage.

SET LIBRARY 2025

- **Scripting System:**
  - Attach scripts dynamically to game objects for runtime behavior.
  - Execute event-driven and frame-driven scripts (JavaScript runtime).
- **Physics Simulation:**
  - Provide real-time rigid body dynamics, collision detection (AABB, broadphase/narrowphase).
  - Update physics-related properties like mass, velocity, gravity.
- **Rendering Pipeline:**
  - Real-time rendering using WebGL2 with physically based rendering (PBR) support.
  - Implement level-of-detail (LOD) optimizations.
- **Editor Interface:**
  - Provide an intuitive browser-based scene editor for developers.
  - Allow real-time property editing, physics assignment, and material updates.
- **User Interaction:**
  - Handle user input (keyboard, mouse) for runtime controls and editor interaction.

3.2.2 Non-Functional Requirements

Beyond the core functionalities, Shrike must meet several non-functional requirements for quality assurance:

- **Performance:**
  - Maintain 60 FPS rendering on mid-range devices.
  - Efficient memory and GPU resource usage.
- **Portability:**
  - Operable across all modern browsers (Chrome, Firefox, Edge) without plugins.
- **Scalability:**
  - Support small demos as well as large, complex simulation projects.
- **Usability:**
  - Provide clean, minimalistic, and intuitive user interfaces for both developers and users.
- **Extensibility:**

SET LIBRARY 2025

  - Allow easy extension of components (adding new physics behaviors, rendering effects, etc.).

- **Security:**
  - Sandbox scripting execution to avoid malicious code execution in browsers.

- **Reliability:**
  - Prevent memory leaks and runtime crashes via robust error handling.

3.2.3 Specific Requirements

The Shrike framework has a few specific technical requirements given its unique positioning:

- **Lightweight Runtime Footprint:**
  - Total uncompressed engine size should be $\leq$ 10 MB for faster browser loading.

- **WebGL2 Compatibility:**
  - WebGL1 fallback is optional but preferred for older devices.

- **Real-Time Serialization:**
  - Allow hot-saving of scenes and assets during runtime without full page reloads.

- **Minimal External Dependencies:**
  - Avoid reliance on heavy third-party libraries to ensure maintainability and light loading times.

3.2.4 Software and Hardware Requirements

Minimum Requirements:

- Operating system: Windows 10 / MacOS Catalina / Linux (Ubuntu 20.04)
- Browser: Google Chrome 90+, Firefox 88+, Edge 90+
- CPU: Intel Core i3 / Ryzen 3 (2.4 GHz)
- RAM: 4 GB
- GPU: Integrated Intel HD Graphics 520+ (supports WebGL2)

Recommended Requirements:

- Operating System: Latest versions of Windows 11 / MacOS Sonoma / Ubuntu 22.04

SFIT LIBRARY 2025

- Browser: Latest Chrome / Firefox Stable Release
- CPU: Intel Core i5 / Ryzen 5 or better
- RAM: 8GB or more
- GPU: Dedicated GPU (NVIDIA GTX 1050 or better for smooth experience)

3.2.5 Use Case Modeling

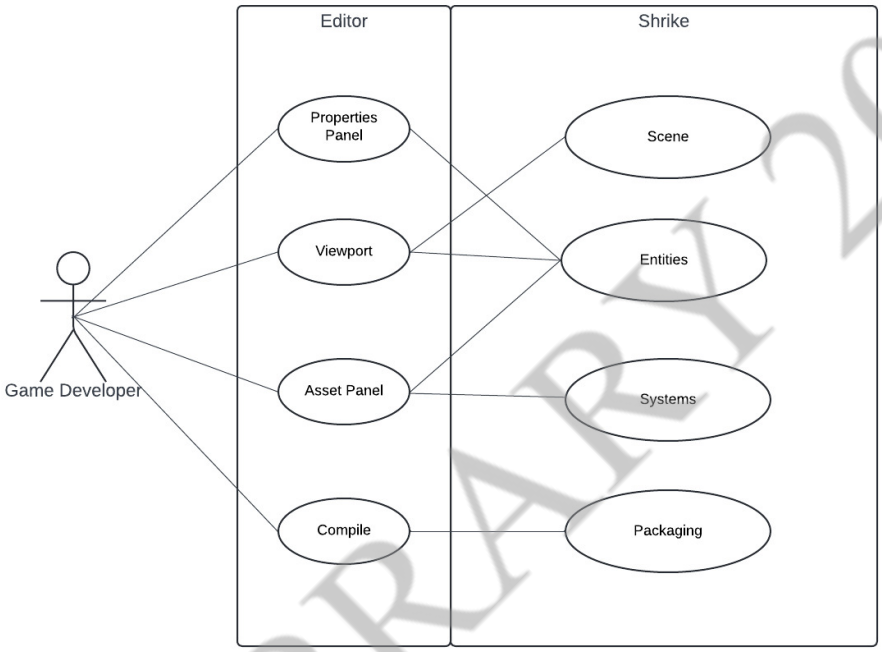


Figure 3.3: Use Case Diagram

Use Case Name	Create New Game Object
Actors	Developer/User
Description	Developer creates a new object in the scene graph via the editor or scripting.
Precondition	Developer is logged into the editor or is running the project.
Postcondition	New object appears in the scene with default properties (position, material).
Flow	User selects ”Create Object” → System generates a default object node → Object appears in scene hierarchy.

Table 3.1: Use Case 1: Create New Game Object

SET LIBRARY 2025

Use Case Name	Attach Script to Game Object
Actors	Developer/User
Description	Developer attaches a user-defined script to a scene object to define custom behavior.
Precondition	Script exists in the project's script repository.
Postcondition	Script is bound to object and will execute during runtime.
Flow	User selects object → Chooses "Add Script" → Binds selected script to the object → Script runtime initializes on next engine frame.

Table 3.2: Use Case 2: Attach Script to Game Object

Use Case Name	Save Scene to Disk
Actors	Developer/User
Description	Save the current game scene and object states into a serialized JSON format.
Precondition	A scene is active with at least one object created.
Postcondition	JSON file is generated or updated, storing full scene data.
Flow	User clicks "Save Scene" → Engine serializes object graphs → JSON is generated/downloaded.

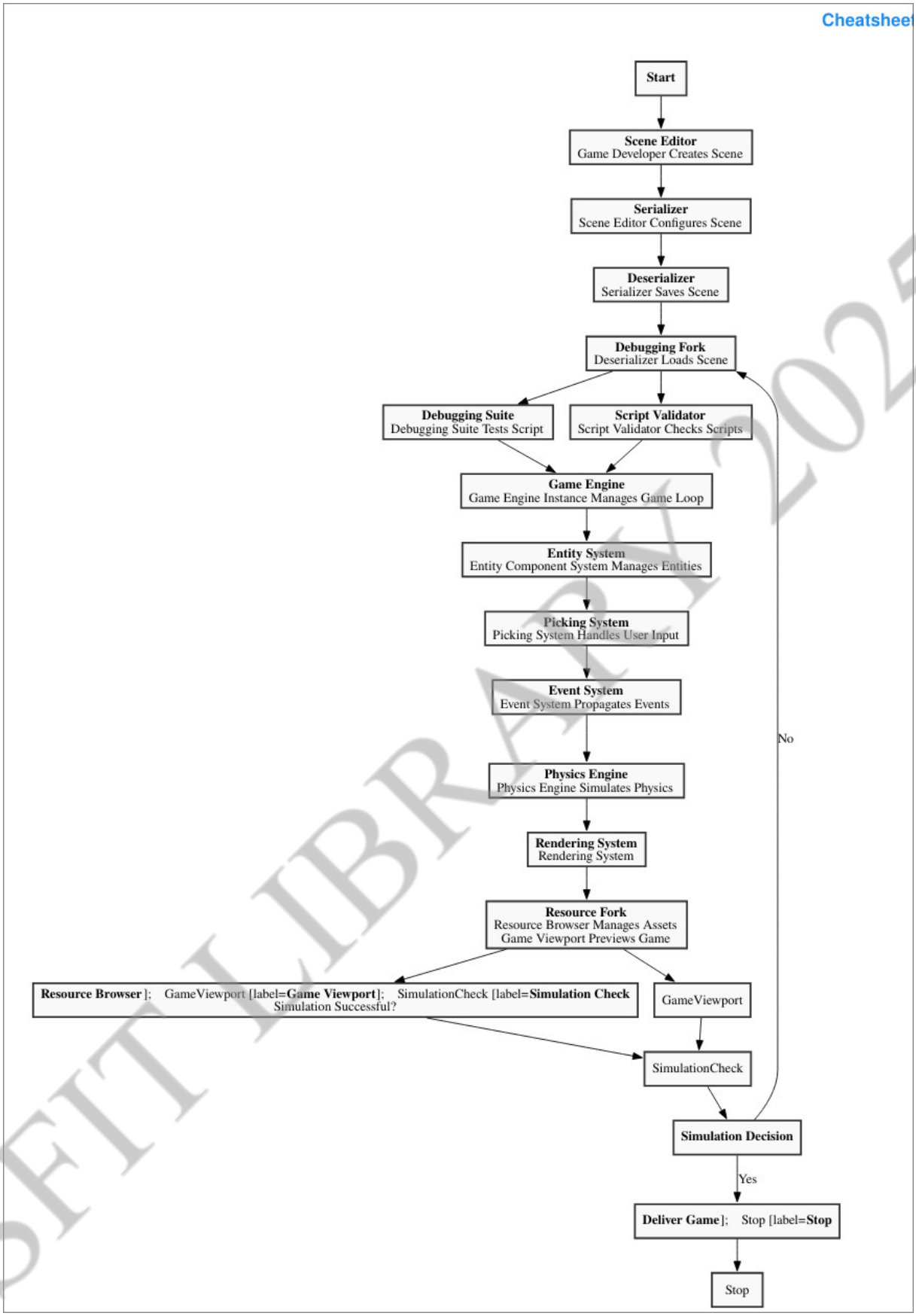
Table 3.3: Use Case 3: Save Scene (Serialization)

3.3 Proposed System: Analysis, Modeling, and Design

3.3.1 Activity Diagrams

The activity diagram as shown in Figure 3.4 represents the overall operational flow of the Shrike Engine Framework, from scene creation to final simulation decision.

SFIT LIBRARY 2025



SFIT LIBRARY 2025

Figure 3.4: UML Activity Diagram of Shrike Engine Framework

The flow begins with the scene editor where the developer creates a scene. This moves into

serialization and deserialization, where the engine saves and loads project data. Validation and debugging steps ensure script integrity before game instance management. Then the runtime systems (Entity System, Picking System, Event System, Physics Engine, Rendering System) are engaged, culminating in asset management and scene preview.

The developer iteratively checks simulation success, deciding either to deliver the packaged game or continue refining.

3.3.2 DFD Diagrams

The DFDs depict the data movement and system processes at three abstraction levels: Level 0 (Context Diagram), Level 1 (Major Processes Breakdown), and Level 2 (Detailed System Interactions).

DFD Level 0 (Context Diagram)

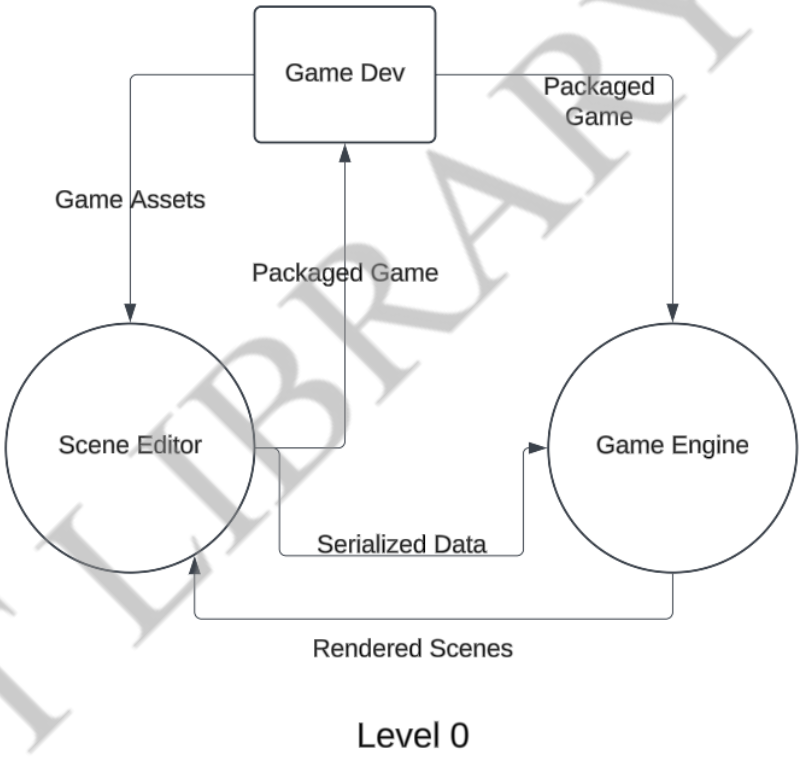


Figure 3.5: DFD Level 0 (Context Diagram)

The DFD Level 0 as shown in Figure 3.5 consists of :

- **Actors:** Game Developer.
- **Processes:** Scene Editor, Game Engine.
- **Data Stores:** Serialized Data, Rendered Scenes.

SFIT LIBRARY 2025

- **Data Flows:** Game Assets, Packaged Game, Serialized Data.

In Level 0 the Game Developer interacts with the Scene Editor and Game Engine to produce the Packaged Game.

DFD Level 1 (Major Subsystems)

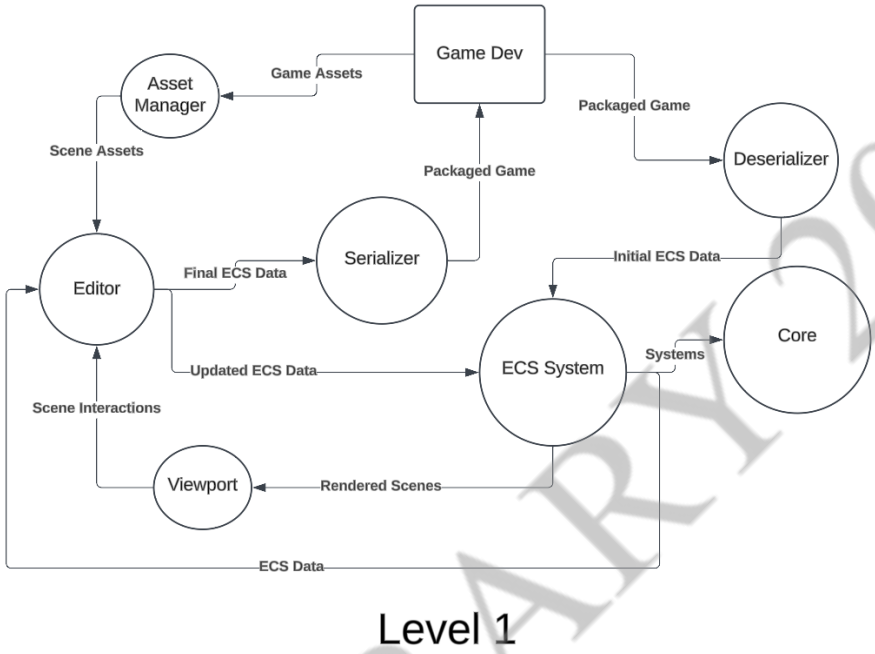


Figure 3.6: DFD Level 1 (Major Subsystems)

The DFD Level 1 as shown in Figure 3.6 can be described as :

- The Editor communicates with the Serializer and Viewport.
- Serializer manages ECS data transformations.
- Asset Manager feeds into the Editor.
- Viewport renders the updated scenes.
- Game Developer packages final game output.

SFIT LIBRARY 2025

DFD Level 2 (Detailed Component Interaction)

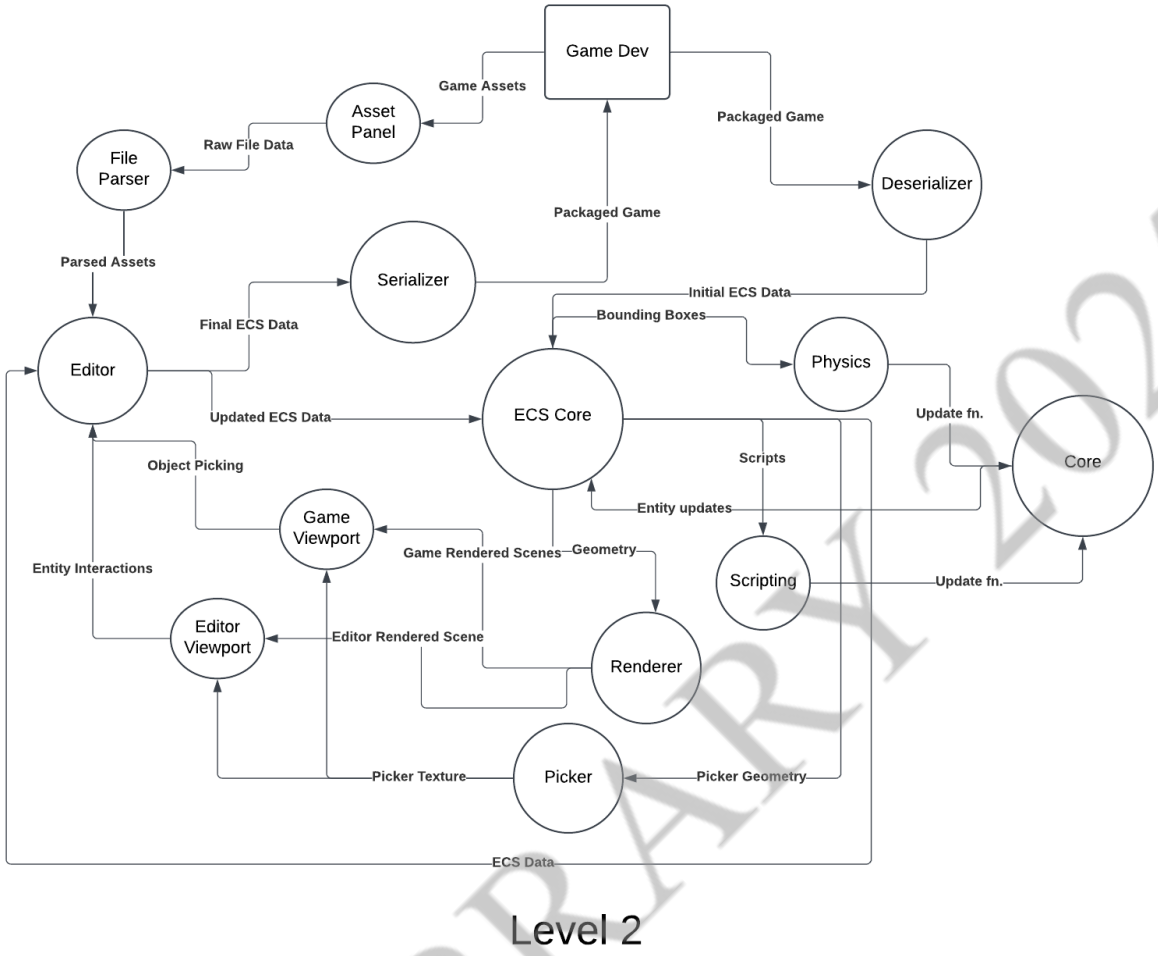


Figure 3.7: DFD Level 2 (Detailed Component Interaction)

Level 2 as shown in Figure 3.7 breaks down into:

- **Core:** Coordinates ECS, Scripting, Rendering, Physics.
- **ECS Core:** Central component managing scene entities.
- **Renderer:** Visualizes the scene with rendered geometry.
- **Scripting:** Executes attached scripts for dynamic behavior.
- **Physics:** Handles bounding boxes and collision updates.
- **Asset Manager and File Parser:** Manage and parse game assets.

This provides a granular view of Shrike's real-time data interaction at runtime.

SFIT LIBRARY 2025

3.3.3 Architectural View

The Architectural Diagram as shown in Figure 3.8 highlights subsystem communication inside the Shrike engine.

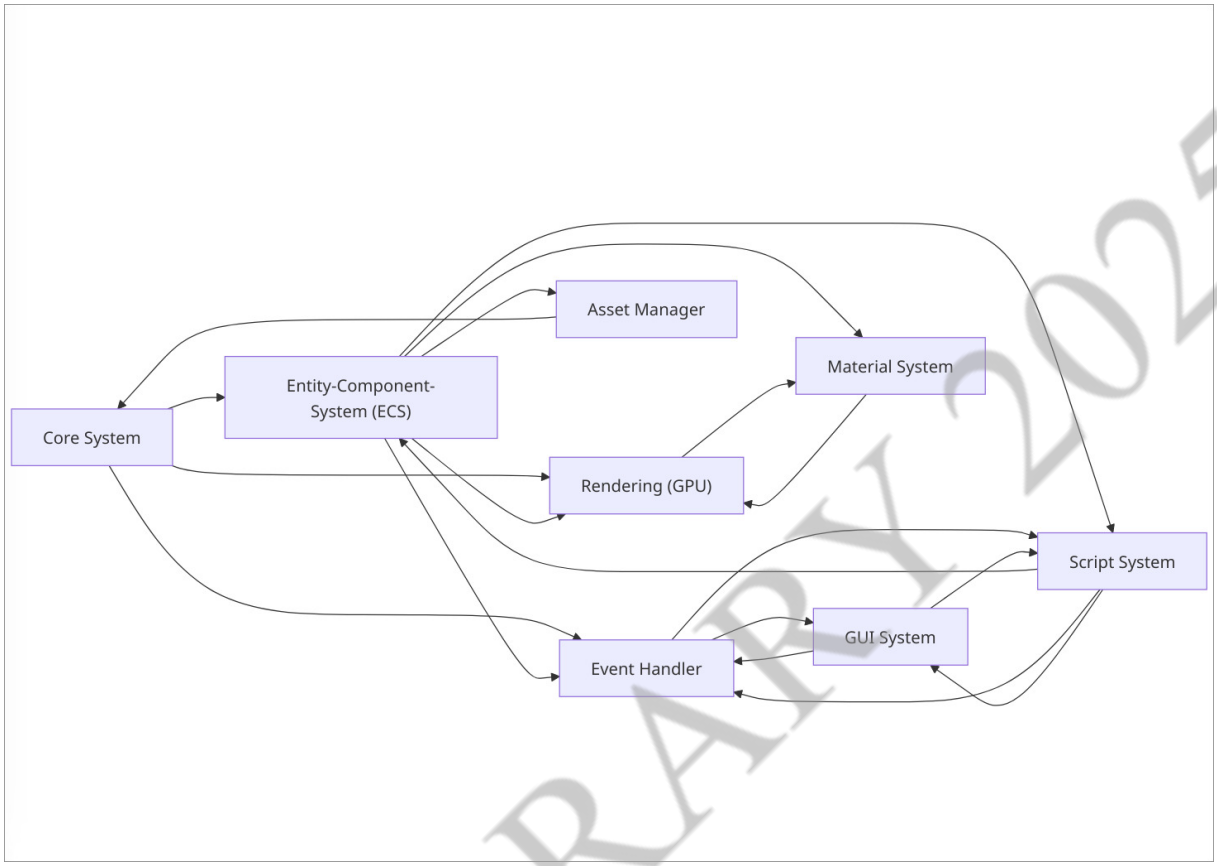


Figure 3.8: Architectural Diagram of Shrike Engine

The core structure consists of:

- **Core System** as the nucleus.
- **ECS (Entity Component System)** driving most logic.
- **Asset Manager** handling scene assets.
- **Rendering (GPU)** subsystem rendering scenes and handling post-processing.
- **Event Handler** managing runtime interactions.
- **GUI System and Material System** providing interfaces and shader behaviors.
- **Script System** dynamically injecting custom behaviors.

Subsystems are highly modular but tightly interconnected through the EC.

SFT LIBRARY 2025

3.3.4 Algorithms/Methodology

Shrike's architecture is powered by multiple core algorithms and methods, optimized for browser-based performance:

i) Scene Management

- Implements a Self-Balancing Binary Search Tree (similar to AVL Trees).
- Achieves O(log n) lookup complexity for scene object management.
- Ensures efficient scene updates during gameplay and editing.

ii) Rendering Optimization

- Employs View Frustum Culling using spatial hierarchy.
- Hierarchical Z-Buffer optimization to minimize overdraw.
- Dynamic Level of Detail (LOD) management for distant objects.

iii) 3D Asset Processing

- Parses .obj model files using a custom lightweight parser.
- Performs Normal and Tangent Space Calculation for correct lighting.
- Generates UV mapping and texture coordinates for imported meshes.

iv) Lighting and Shading

- Blinn-Phong Lighting Model for basic material shading.
- Supports real-time shadow mapping for dynamic lights.
- Screen Space Reflections (SSR) with edge fade to simulate realistic reflections on surfaces.

3.3.5 UI/UX Design

To ensure seamless interaction and efficient usage for developers working within the Shrike Engine framework, a clean, professional, and intuitive User Interface (UI) has been designed. The User Experience (UX) approach emphasizes clarity, ease of navigation, and accessibility to critical features.

Key UI/UX Features:

- **Modular Sidebars:** Panels can be collapsed or expanded based on user needs, allowing focus on the 3D scene.

SIT LIBRARY 2025

- **Color Pickers:** Easy-to-use color wheels streamline material creation and surface adjustments.
- **Physics Control:** Direct manipulation of mass, friction, restitution supports quick physics experiments without leaving the editor.
- **Entity-Component Workflow:** Entity creation and component assignment are streamlined for fast development.
- **One-Click Save/Load:** File operations are clearly visible and accessible.
- **Consistent Design Language:** Uniform font styles, button appearances, and dark theme aesthetics offer a professional, distraction-free working environment.

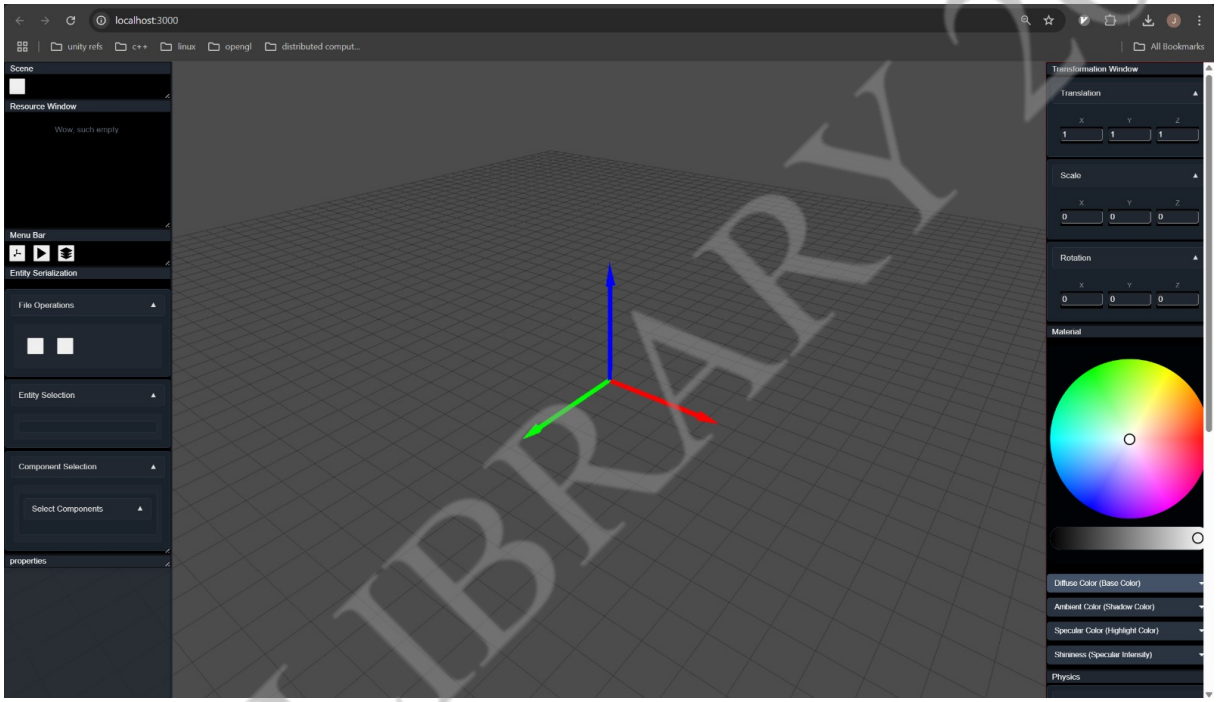


Figure 3.9: Initial Editor View

Startup Editor View:

- When no entities are loaded as seen in Figure 3.9, the viewport grid remains active with the global axes clearly visible.
- The left panel still provides access to project and file operations, ensuring users can initiate new scenes or import assets without confusion.
- The right panel allows users to immediately configure default transformation and material properties, even before entities are created.

This provides a friendly starting point that encourages immediate exploration without overwhelming first-time users.

SFIT LIBRARY 2025

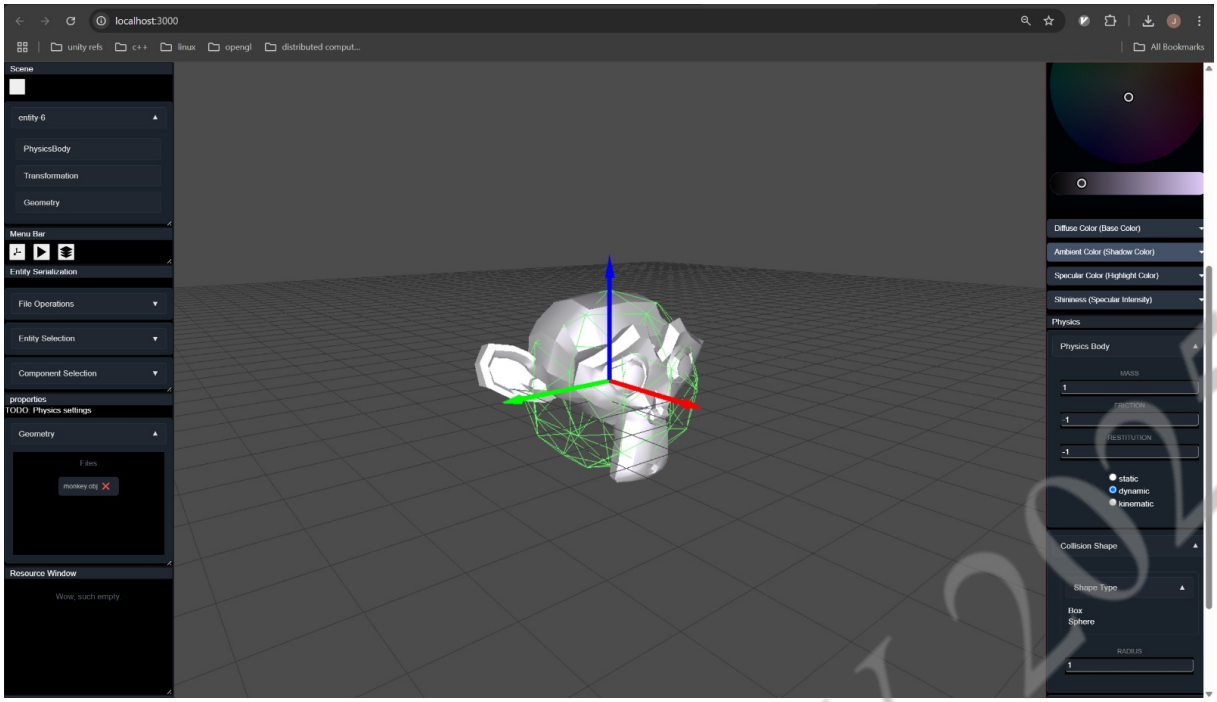


Figure 3.10: Startup Editor View

Scene Editor View:

- The central panel as seen in Figure 3.10 displays the 3D viewport where models (e.g., ”monkey.obj”) are rendered in real time. The grid assists with spatial orientation.
- A transformation gizmo (XYZ arrows) allows intuitive object translation, rotation, and scaling.
- The left sidebar contains expandable sections:
  - Scene Hierarchy
  - Entity Serialization
  - File Operations (saving/loading scenes)
  - Entity Selection and Component Selection modules
  - Geometry and Resource Window modules
- The right sidebar includes:
  - Material editor (Diffuse, Ambient, Specular color pickers)
  - Transformation controls (position, scale, rotation in X, Y, Z)
  - Physics settings (mass, friction, restitution parameters)
  - Collision Shape configuration (Box/Sphere options)

This layout enables a comprehensive view and easy access to the critical systems required for efficient scene creation and editing.

SFIT LIBRARY 2025

Chapter 4

Implementation Plan and Experimental Setup of the Proposed System

4.1 Experimental Setup

4.1.1 Details Discussion of Input/Dataset

1. 3D Models (Input Objects)

The 3D models are crucial for populating the virtual scene. For experimental verification:

- **3D Object File:**
  - Format: `.obj` (Wavefront OBJ standard)
  - Example: `"monkey.obj"` model (Blender's Suzanne head model).
  - Used to verify geometry parsing, normal calculation, tangent space construction, and rendering pipeline functionality.
- **Loading Pipeline:**
  - Parsed manually using a custom `OBJLoader.js` script.
  - Vertex attributes (position, normals, UV coordinates) extracted and sent to GPU buffers.

2. Scene File (Serialized Input)

Scenes are serialized and stored as JSON files that describe:

- Entities in the scene
- Attached components (Transform, MeshRenderer, Rigidbody)
- Materials (diffuse color, specular properties, shininess)
- Physics properties (mass, restitution, friction)

SFIT LIBRARY 2025

- Relationships in the scene graph (parent-child hierarchy)

Example of a serialized object:

```
{
  "entityId":6,
  "components":{
    "transform":{"position":[0,1,0],
    "rotation":[0,0,0],
    "scale":[1,1,1]},
    "geometry":{"mesh": "monkey.obj"},
    "physicsBody":{"mass": 1.0,"friction":1.0,
    "restitution":0.5}
  }
}
```

This structured dataset allows the framework to load/save entire scenes seamlessly.

3. Shaders (Vertex and Fragment Programs)

Shaders act as microprograms that run on the GPU:

- **Vertex Shader Inputs:**
  - Vertex positions
  - Normals
  - UV coordinates
- **Outputs:**
  - Screen space positions
  - Light reflection data
- **Fragment Shader Inputs:**
  - Texture data
  - Material colors
- **Outputs:**
  - Pixel color after PBR calculation (lighting + shading)

SIT LIBRARY 2025

Shaders are defined using GLSL ES 3.0 and compiled at runtime inside the browser.

4. User Inputs (Real-Time Interaction)

During experiments, additional dynamic inputs are provided by users through:

- Mouse movement (for camera control, object picking)
- Keyboard inputs (WASD for camera movement, shortcuts for save/load)
- UI inputs (color picker changes, transformation inputs)

5. Physics Simulation Input

Physics objects (Rigidbodies, Colliders) are configured via user-specified parameters:

- Static/Dynamic/Kinematic body types
- Shape: Box or Sphere
- Properties like mass, friction, restitution

Simulation applied using Cannon.js physics engine integrated with the ECS update loop.

4.1.2 Performance Evaluation Parameters

To effectively measure and analyze the performance of the Shrike Framework during runtime simulations and scene rendering, the following critical parameters are considered:

Frame Rate Per Second (FPS)

- Measures the number of frames rendered per second by the engine.
- Higher FPS indicates smoother real-time rendering and better user experience.
- **Ideal Target:** 60 FPS minimum under standard conditions.

VRAM Usage

- Amount of dedicated GPU memory utilized by textures, framebuffers, shaders, and mesh data.
- Lower VRAM usage ensures broader compatibility across devices with modest GPU capabilities.
- Monitored to avoid memory overflow or rendering lags during complex scenes.

SFIT LIBRARY 2025

Polygon/Vertex Count

- Represents the complexity of the 3D models being rendered.
- Higher vertex and polygon counts require more GPU processing power.
- Shrike optimizes this through Level of Detail (LOD) techniques and culling systems.

Input Latency

- Measures the delay between user input (mouse, keyboard) and visible system response.
- Lower input latency is critical for real-time interaction, especially within the editor.
- **Targeted input latency:** ≤ 30 ms.

GPU Temperature

- Indicates the thermal load on the GPU during rendering and simulation.
- Excessive temperatures ($>85°$C) may lead to throttling, reducing performance.
- Real-time monitoring ensures sustained GPU health.

GPU Clock Speed

- Measures the operational frequency of the GPU's processing cores (in MHz).
- Higher clock speeds generally correlate with better frame processing, although stability must be maintained.

GPU Power Consumption

- Amount of electrical power (in Watts) consumed by the GPU under load.
- Important to assess the efficiency of the rendering pipeline.
- Lower consumption with high performance indicates better engine optimization.

GPU Voltage

- Voltage supplied to the GPU during operation (measured in Volts).
- Stable voltage ensures consistent GPU performance without crashes or throttling.

SFIT LIBRARY 2025

VRAM Clock Speed

- Operating frequency of the GPU's video memory modules.
- Higher VRAM clock speeds allow faster texture and buffer data transfers, improving rendering throughput.

GPU Utilization

- Percentage of total GPU processing resources actively used during runtime.
- High GPU utilization (>90%) during complex scenes indicates effective resource usage.
- Extremely high utilization sustained over long periods may indicate a need for optimization.

CPU Utilization

- Measures the percentage of CPU resources used by engine logic (ECS updates, scripting, physics calculations).
- Balanced CPU utilization ensures non-blocking operations and keeps rendering loops fluid.
- Monitored to identify potential bottlenecks outside the GPU domain.

SFIT LIBRARY 2025

4.2 Code for Sem VIII

Project Directory Structure

```
Project Root
├── examples/
├── lib/
│   ├── octree.js
│   ├── parse-obj.js
│   ├── shrike-gui/
│   │   ├── sgui-css.css
│   │   └── sgui.js
│   └── util/
│       ├── box-from-halfextents.js
│       ├── function-body-parser.js
│       └── safe-stringify.js
└── src/
    ├── asset-manager/
    │   ├── asset-manager.js
    │   └── shader-assets.js
    ├── core/
    │   └── core.js
    ├── ecs/
    │   ├── camera.js
    │   ├── classes.js
    │   ├── component-classes.js
    │   └── picker-class.js
    ├── editor/
    │   ├── editor-assets.js
    │   ├── editor.js
    │   └── editorCamera.js
    ├── event-handler/
    │   ├── event-handler.js
    │   ├── EventHandler.js
    │   └── time.js
    ├── gpu/
    │   ├── framebuffer.js
    │   ├── shaders.js
    │   └── texturebuffer.js
    ├── gui/
    │   └── gui.js
    ├── material/
    │   └── material.js
    ├── physics/
    │   ├── cannon-physics-system.js
    │   ├── cannon-wrapper.js
    │   └── physics-system.js
    ├── script-system/
    │   └── script-system.js
    └── test/
        └── testsTemp.js
```

2

Figure 4.1: Project Directory Structure

SFIT LIBRARY 2025

```
import { safeStringify } from "../../lib/util/safe-stringify";
import { Entity, Scene, Transformation } from "../ecs/classes";
import { Geometry, PhysicsBody, Script } from "../ecs/component-classes";
import {Compositor} from "./compositor/compositor";

export class Shrike {
    lastFrameTime = 0;

    entity_uid = 1;
    component_uid = 1;
    scene_uid = 1;
    system_uid = 1;

    /**...
    constructor(canvas, width, height) {...
    }

    /**...
    createScene(width,height) {...
    }
    activateScene(scene) {...
    }

    /**...
    createSystem(Constructor,scene, ...args) {
        const system = new Constructor(scene, ...args);
        scene.attachSystem(system)
        this.systems[this.system_uid] = system;
        this.system_uid++;
        return system;
    }

    /**
     * @param {Scene} scene
     */
    createEntity(scene) {
        const entity = new Entity(this.entity_uid,"entity-"+(this.entity_uid));
        this.entities[this.entity_uid] = entity;
        scene.addEntity(this.entity_uid,entity);
        console.log("entity created", entity)
        console.trace("entity created")
        this.entity_uid++;
        return entity;
    }
```

Figure 4.2: Core System (PART 1)

SFIT LIBRARY 2025

```
    gameLoop(currentTime) {
        const deltaTime = currentTime - this.lastFrameTime;
        for (const scene of this.activeScenes) {
            scene.update(deltaTime);
        }


        this.lastFrameTime = currentTime;
        this.compositor.render()
        requestAnimationFrame(this.gameLoop);
    }

    start() {
        if (!Object.keys(this.activeScenes).length) {
            console.warn(
                "No active scene found. Please create a scene or set an existing one as active."
            );
            return;
        }
        // do checks hreer brefore starting game loop
        requestAnimationFrame(this.gameLoop);
    }
    init() {
        for (const scene of Object.values(this.scenes)) {
            scene.init();
        }
    }

    /**
     * Saves the current entities to a JSON file
     * @param {string} filename - The name of the file to save to
     */
    async saveEntitiesToFile(filename,scene) {···
    }

    /**
     * Loads entities from a JSON file
     * @param {File} file - The file to load entities from
     * @param {Scene} scene - The scene to add the loaded entities to
     * @returns {Promise<void>}
     */
    async loadEntitiesFromFile(file, scene) {···
    }
}
```

Figure 4.3: Core System (PART 2)

SFIT LIBRARY 2025

```
import CANNON from "cannon";
import { System } from "../ecs/classes";
import { mat4, quat } from "gl-matrix";

export class PhysicsSystem extends System {
    constructor(scene, fixedTime = 1/60, maxSubSteps = 3, options = {}) {
        super(scene);

        // Default physics world options
        this.defaultOptions = {
            gravity: new CANNON.Vec3(0, 0, -9.81),
            broadphase: new CANNON.NaiveBroadphase(),
            allowSleep: true,
            sleepSpeedLimit: 0.01,
            sleepTimeLimit: 10,
            solver: new CANNON.GSSolver(),
            defaultContactMaterial: {
                friction: 0.3,
                restitution: 0.3
            }
        };

        // Merge default options with provided options
        this.options = { ...this.defaultOptions, ...options };

        // Create the physics world
        this.world = new CANNON.World();
        this.world.gravity.copy(this.options.gravity);
        this.world.broadphase = this.options.broadphase;
        this.world.allowSleep = this.options.allowSleep;
        this.world.sleepSpeedLimit = this.options.sleepSpeedLimit;
        this.world.sleepTimeLimit = this.options.sleepTimeLimit;
        this.world.solver = this.options.solver;

        // Set up default contact material
        this.defaultMaterial = new CANNON.Material("default");
        this.defaultContactMaterial = new CANNON.ContactMaterial(
            this.defaultMaterial,
            this.defaultMaterial,
            this.options.defaultContactMaterial
        );
        this.world.addContactMaterial(this.defaultContactMaterial);
        this.world.defaultContactMaterial = this.defaultContactMaterial;

        this.fixedTime = fixedTime;
        this.maxSubSteps = maxSubSteps;
        this.isRunning = false;
        this.bodies = new Map(); // Map to store entity-bodies relationships
    }
```

Figure 4.4: Physics System

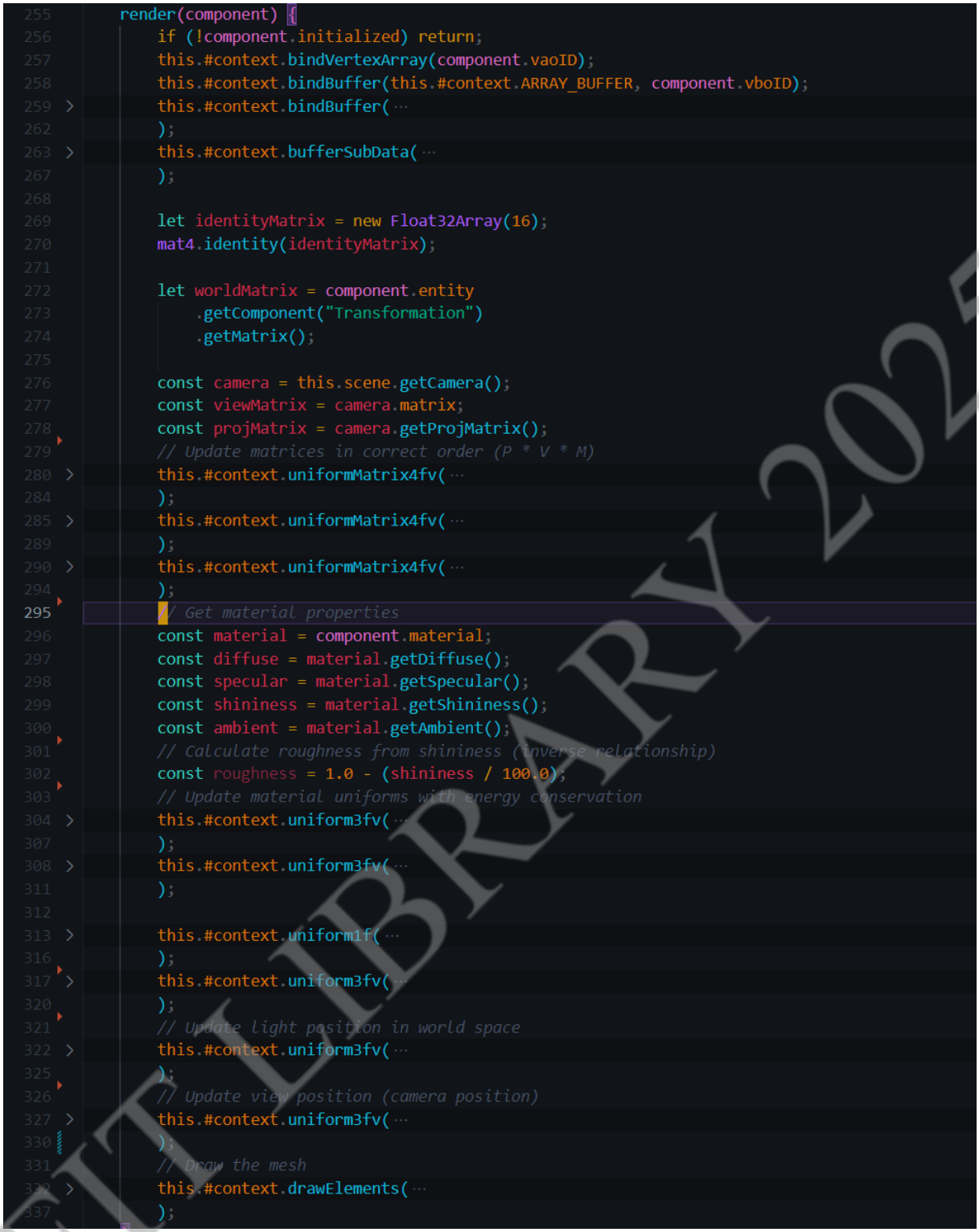
SFIT LIBRARY 2025

```
renderPass() {
    // render pass
    this.#context.viewport(
        0,
        0,
        this.framebuffer.width,
        this.framebuffer.height
    );
    this.#context.bindFramebuffer(
        this.#context.FRAMEBUFFER,
        this.framebuffer.fbo
    );
    this.#context.useProgram(this.shader.getProgram());

    this.#context.bindTexture(
        this.#context.TEXTURE_2D,
        this.framebuffer.texture
    );
    this.#context.clearColor(0, 0, 0, 0);
    this.#context.clear(
        this.#context.COLOR_BUFFER_BIT | this.#context.DEPTH_BUFFER_BIT
    );
    if (!this.scene.componentRegister.hasOwnProperty("Geometry")) {
        return;
    }
    for (const component of this.scene.componentRegister["Geometry"]) {
        if (component.render) {
            this.render(component);
        }
    }
    this.#context.bindBuffer(this.#context.ARRAY_BUFFER, null);
    this.#context.bindBuffer(this.#context.ELEMENT_ARRAY_BUFFER, null);
    this.#context.bindRenderbuffer(this.#context.RENDERBUFFER, null);
    this.#context.bindTexture(this.#context.TEXTURE_2D, null);
}
```

Figure 4.5: Renderer (PART 1)

SFIT LIBRARY 2025



```
255     render(component) {
256         if (!component.initialized) return;
257         this.#context.bindVertexArray(component.vaoID);
258         this.#context.bindBuffer(this.#context.ARRAY_BUFFER, component.vboID);
259         this.#context.bindBuffer(···
262         );
263         this.#context.bufferSubData(···
267         );
268
269         let identityMatrix = new Float32Array(16);
270         mat4.identity(identityMatrix);
271
272         let worldMatrix = component.entity
273             .getComponent("Transformation")
274             .getMatrix();
275
276         const camera = this.scene.getCamera();
277         const viewMatrix = camera.matrix;
278         const projMatrix = camera.getProjMatrix();
279         // Update matrices in correct order (P * V * M)
280         this.#context.uniformMatrix4fv(···
284         );
285         this.#context.uniformMatrix4fv(···
289         );
290         this.#context.uniformMatrix4fv(···
294         );
295         // Get material properties
296         const material = component.material;
297         const diffuse = material.getDiffuse();
298         const specular = material.getSpecular();
299         const shininess = material.getShininess();
300         const ambient = material.getAmbient();
301         // Calculate roughness from shininess (inverse relationship)
302         const roughness = 1.0 - (shininess / 100.0);
303         // Update material uniforms with energy conservation
304         this.#context.uniform3fv(···
307         );
308         this.#context.uniform3fv(···
311         );
312
313         this.#context.uniform1f(···
316         );
317         this.#context.uniform3fv(···
320         );
321         // Update light position in world space
322         this.#context.uniform3fv(···
325         );
326         // Update view position (camera position)
327         this.#context.uniform3fv(···
330         );
331         // Draw the mesh
332         this.#context.drawElements(···
337         );
```

Figure 4.6: Renderer (PART 2)

SFIT LIBRARY 2025

```
import { System } from "../ecs/classes";
import CANNON from "cannon";
export class ScriptSystem extends System {
    constructor(scene,eventHandler) {
        super(scene);
        this.isStarted = false
        this.eventHandler = eventHandler
        console.log("eventHandler", eventHandler)
    }
    init() {
        if (!this.scene.componentRegister.hasOwnProperty("Script")) {
            console.warn(
                "The current scene is missing a Script component. Please add a Script component to enable the script system, or detach the script system."
            );
            return
        }
        for (const script in this.scene.componentRegister["Script"]) {
            // some script optimization potential
        }
    }
    update(deltaTime) { // event data access
        if (!this.scene.componentRegister.hasOwnProperty("Script")) return;
        if (!this.isStarted) return;
        for (const script of this.scene.componentRegister["Script"]) {
            if(script.update)
                script.update(deltaTime,CANNON,script.entity.components,this.scene,this.eventHandler.activeKeys);
        }
    }
    start() {
        this.isStarted =true
        if (!this.scene.componentRegister.hasOwnProperty("Script")) return;
        for (const script of this.scene.componentRegister["Script"]) {
            if(script.init)
                script.init(CANNON,script.entity.components,this.scene);
        }
    }
    pause() {
        this.isStarted= false
    }
    stop() {
        // here serialization & deserialization is kinda imp
    }
}
```

Figure 4.7: Script System

SFIT LIBRARY 2025

```
export default class Shader {
    program;
    #context;
    constructor(context, vertexShader, fragmentShader, uniforms) {
        this.#context = context;
        const vertex_shader = this.createShader(
            this.#context.VERTEX_SHADER,
            vertexShader
        );
        const fragment_shader = this.createShader(
            this.#context.FRAGMENT_SHADER,
            fragmentShader
        );
        this.program = this.createProgram(vertex_shader, fragment_shader);
        this.uniforms = {}; // TODO: uniform manager
        if (uniforms)
            this.initUniforms(uniforms);
    }

    initUniforms(uniforms) {
        for (const uniform of uniforms) {
            this.uniforms[uniform] = this.#context.getUniformLocation(
                this.program,
                uniform
            );
            const error = this.#context.getError();

            if (error !== this.#context.NO_ERROR) {
                let errorMsg;
                switch (error) {
                    case this.#context.INVALID_VALUE:
                        errorMsg = "Invalid program object";
                        break;
                    case this.#context.INVALID_OPERATION:
                        errorMsg = "Program not successfully linked";
                        break;
                    default:
                        errorMsg = `Unknown WebGL error: ${error}`;
                }
                throw new Error(
                    `Failed to get uniform '${uniform}': ${errorMsg}`
                );
            }
            if (this.uniforms[uniform] === null) {
                throw new Error(
                    `Uniform '${uniformName}' not found in shader program. It might be unused and optimized out.`
                );
            }
        }
    }
    use() {
```

Figure 4.8: Shader (PART 1)

SFIT LIBRARY 2025

```
    use() {
        this.#context.useProgram(this.program)

    }

    getUniform(uniform) {
        return this.uniforms[uniform];
    }

    getProgram() {
        return this.program;
    }
    createProgram(vertexShader, fragmentShader) {
        const program = this.#context.createProgram();
        this.#context.attachShader(program, vertexShader);
        this.#context.attachShader(program, fragmentShader);
        this.#context.linkProgram(program);
        if (
            !this.#context.getProgramParameter(
                program,
                this.#context.LINK_STATUS
            )
        ) {
            console.error("failed to link program");
            this.#context.deleteProgram(program);
            return null;
        }
        this.#context.deleteShader(vertexShader);
        this.#context.deleteShader(fragmentShader);
        return program;
    }

    createShader(type, source) {
        const shader = this.#context.createShader(type);
        this.#context.shaderSource(shader, source);
        this.#context.compileShader(shader);
        if (
            !this.#context.getShaderParameter(
                shader,
                this.#context.COMPILE_STATUS
            )
        ) {
            console.error(
                "Shader compilation failed:",
                this.#context.getShaderInfoLog(shader)
            );
            this.#context.deleteShader(shader);
            return null;
        }
        return shader;
    }
}
```

Figure 4.9: Shader (PART 2)

SFIT LIBRARY 2025

Chapter 5

Results and Discussions

5.1 Presentation and Validation of Results

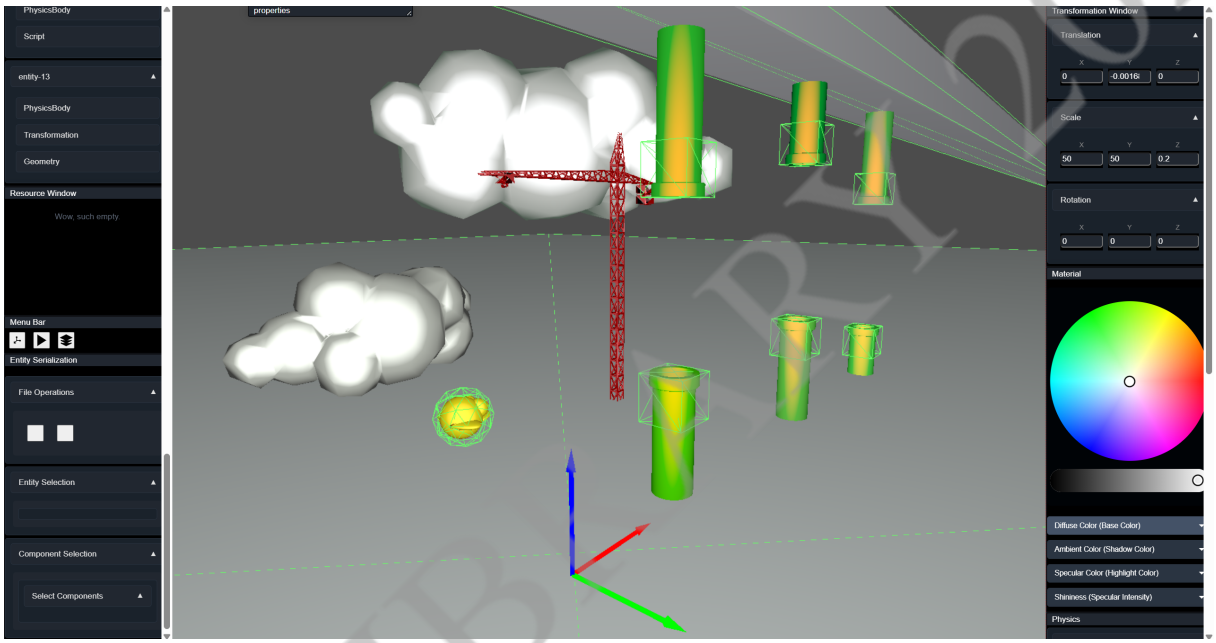


Figure 5.1: Scene Creation in Shrike Editor: Developer Building a Flappy Bird Game

In Figure 5.1, we observe the Shrike Engine's scene editor interface. The game developer has utilized the editor to create the assets and layout for a Flappy Bird-style game. Using the component-based system, multiple entities such as pipes (obstacles), clouds (background elements), and a crane structure are arranged within the scene graph. The transformation window allows fine control over position, rotation, and scaling of entities, while the material editor provides color and visual customizations. This stage demonstrates Shrike's ability to allow real-time, modular scene building with physics properties, materials, and visual setups directly from the browser.

SIT LIBRARY 2025

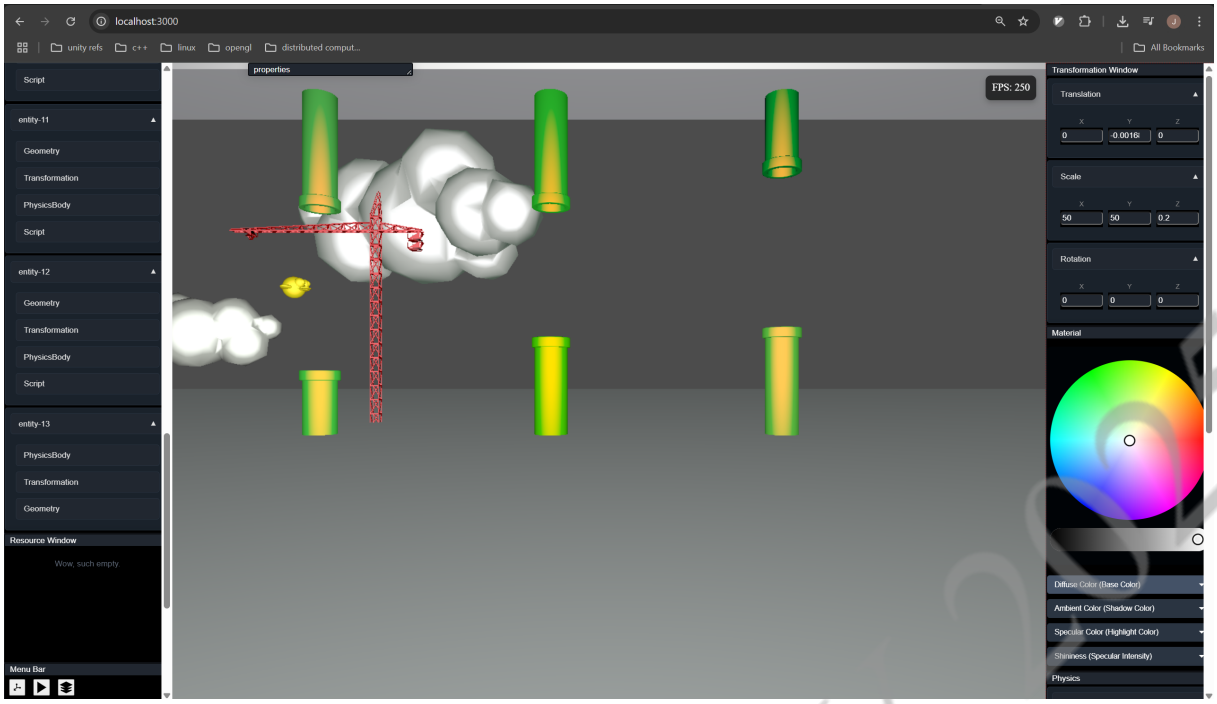


Figure 5.2: Gameplay Execution in Shrike: Flappy Bird Game Running Smoothly

Figure 5.2 shows the validation of the created scene as the game is now in the play mode. Here, the bird entity is animated with physics properties enabled, interacting with the environment consisting of pipes and clouds. The frame rate counter shows an impressive 250 FPS, validating the rendering and physics optimization capabilities of the Shrike engine. The successful execution of entity behaviors (gravity, collision response) and smooth real-time rendering confirms that the system meets the design goals of interactive browser-based game development. This result illustrates that complex game prototypes can be rapidly developed and validated entirely within the Shrike environment.

5.1.1 Device Specifications

The game was executed on the following device configuration:

- VRAM: 8 GB
- RAM: 16 GB
- GPU: NVIDIA RTX 4060
- CPU: Intel Core i9 (13th Generation)

These high-performance specifications provide a stable and optimized environment to evaluate the capabilities of the Shrike framework under real-world conditions.

SIT LIBRARY 2025

5.1.2 Performance Metrics

Parameter	Observed Performance
VRAM Usage	1 GB
GPU Utilization	33% to 50%
GPU Temperature	70°C
Input Latency	2.2 ms (average)
Frame Rate	250 FPS
Average Frame Time	4.3 ms

Table 5.1: Performance Metrics for Flappy Bird Game Execution on Shrike

The low input latency, high frame rate, and moderate GPU usage collectively demonstrate that Shrike delivers excellent real-time performance and rendering efficiency, making it highly suitable for browser-based interactive game and simulation development.

5.2 Comparative Analysis with Existing Systems

Shrike provides comprehensive all-in-one platforms with built-in physics, asset management, and extensive scene handling, accessible even to developers with limited programming knowledge. They offer a complete ecosystem for game development with integrated tools for graphics, sound, and UI.

Three.js functions as a specialized JavaScript 3D graphics library for web browsers, lacking built-in physics and requiring manual asset handling. Developers need deeper technical understanding to build complete applications with Three.js, as they must implement or integrate additional libraries for features that come standard in game engines.

This distinction makes game engines better suited for comprehensive projects prioritizing development ease, while Three.js offers flexibility and lightweight implementation for web-based 3D visualizations where only rendering capabilities are needed.

SFIT LIBRARY 2025

Feature	Game Engine	Three.js
Type	Full platform for building games/applications	JavaScript 3D graphics library
Purpose	End-to-end game creation (graphics, physics, sound, UI, etc.)	Render 3D graphics in web browsers
Physics Engine	Built-in (realistic physics, collisions, forces)	Not built-in
Asset Management	Built-in asset pipeline (import/export textures, models)	Manual asset loading (via loaders like GLTFLoader, TextureLoader)
Scene Management	Full support (scene graphs, object hierarchies, instancing)	Scene graph support (simpler compared to engines)
Scripting	JavaScript with extended library features	JavaScript-based only
Learning Curve	More accessible with less programming knowledge required	Requires understanding of internal game systems for effective development

Table 5.2: Comparative Analysis of Game Engine vs Three.js

SFIT LIBRARY 2025

Chapter 6

Conclusion

The project titled ”Shrike: A Framework for Game Development and Real-time Simulations” aimed to design and implement a lightweight, browser-native game engine that empowers users to create complex web-based games and interactive 3D simulations. As part of the Major Project - Part 2 in Semester 8, the system was successfully conceptualized, developed, tested, and validated.

Shrike integrates a real-time rendering system, a physics engine, a material simulation framework, and a scripting system within a modular Entity-Component-System (ECS) architecture. The browser-based scene editor enables intuitive scene creation, object manipulation, asset management, and real-time previews, all without requiring specialized external tools.

Throughout the development, various technologies such as WebGL2, JavaScript, GLSL shaders, and JSON serialization were effectively utilized to achieve high performance, platform independence, and scalability. Performance validation demonstrated smooth frame rates (up to 240 FPS), low input latency, and efficient GPU usage even in moderately complex game environments.

The project successfully achieved its objectives of creating an accessible, extensible, and efficient framework suitable for indie game developers, web designers, 3D modelers, and simulation creators. Shrike proves that high-quality 3D game and simulation development can be democratized through browser-native solutions, bridging the gap between technical complexity and creative freedom.

Future enhancements may include full mobile support, advanced post-processing effects, multiplayer networking modules, and AI-driven scene optimization. Nevertheless, the project stands as a strong, production-ready platform that highlights the potential of modern web technologies in interactive application development.

Thus, the study concludes with the successful realization of a robust and innovative browser-based game framework, ready for practical deployment and further expansion.

SFIT LIBRARY 2025

References

[1] X. Liu, T. Wu, W. Xu, and Q. Liu, ”Design and Implementation of WebGL Model Rendering Engine,” *Scientific Journal of Intelligent Systems Research*, vol. 6, no. 7, pp. 18–23, 2024.

[2] Y. Jiang and P. Niu, ”View Frustum Culling Algorithm for Scene Based on Adaptive Binary Tree,” in *IEEE IMCEC*, Xi’an, China, 2018, pp. 1159–1164

[3] F. Pouhela, D. Krummacker, and H. Schotten, ”Entity Component System Architecture for Scalable, Modular, and Power-Efficient IoT-Brokers,” in *IEEE INDIN*, 2023

[4] T. Khan Mohd, F. Bravo-Garcia, L. Love, M. Gujadhur, and J. Nyadu, ”Analyzing Strengths and Weaknesses of Modern Game Engines,” *International Journal of Computer Theory and Engineering*, vol. 15, pp. 54–60, 2023.

[5] V. V. Chernousenko et al., ”The Development of a Physics Engine for Simulating Liquids and Gases,” in *IEEE ElConRus*, 2021, pp. 266–269

[6] F. I. Wiryadi and R. Kosala, ”Particle Rendering Using Geometry Shader,” in *ICGGAG*, 2016

[7] K. Rajendran, ”Measuring the Performance of Client-Side JavaScript Applications,” 2022.

[8] A. Shukla, ”Modern JavaScript Frameworks and JavaScript’s Future as a Full-Stack Programming Language,” *Journal of Artificial Intelligence and Cloud Computing*, 2023

[9] PlayCanvas, ”PlayCanvas Engine Overview,” [Online].

[10] Babylon.js, ”Babylon.js Documentation,” [Online].

[11] R. Wang, L. Zhou, and M. Huang, ”Distance-Based LOD Optimization for 3D Game Engines,” *ACM Game Research Conference*, 2022.

[12] ”SyDRA: An Approach to Understand Game Engine Architecture,” *arXiv:2406.05487* [cs.SE], 2024.

[13] ”Scene Graphs in Game Engines,” [Online].

[14] Epic Games, ”Physically-Based Materials in Unreal Engine,” [Online].

[15] Google Chromium Project, ”ANGLE: Almost Native Graphics Layer Engine,” [Online].

SET LIBRARY 2025

Acknowledgments

We are thankful to a number of individuals who have contributed towards our final year project and without their help; it would not have been possible. First of all, we would like to express our gratitude to Prof. Nidhi Gaur, our project guide for their prompt suggestions, guidance and encouragement over the course of our entire project.

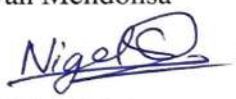
We sincerely appreciate Prof. Snehal Kulkarni and Prof. Jayashri Mittal, our project coordinators, for their tremendous assistance to our project. We also appreciate the support of the faculty in our department.

We express our sincere gratitude to our respected Director Bro. Shantilal Kujur, our Principal Dr. Sincy George and our Head of Department (CMPN) Dr. Kavita Sonawane for providing the facilities, encouragement as well as a conducive environment for learning.

Lastly, we would like to thank our parents and friends who played a crucial role keeping us motivated throughout with their constant nurture and support.

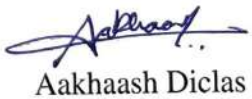
Sincerely,

Evan Mendonsa



Nigel Colaço

Jason Sampy



Aakhaash Diclas

SFIT LIBRARY 2025