FlockJS: A Browser-Native Game Engine Integrating WebGPU and Peer-to-Peer Networking for Scalable Multiplayer Experiences

Faris Jiwad, Owen Wolff, Omar Barabandi, Laith Najjab, and Xiaodong Qu[0000−0001−7610−6475]

The George Washington University

Abstract. FlockJS is a lightweight, browser-native game engine that enables scalable, real-time multiplayer gameplay without centralized servers. By integrating WebGPU for GPU-accelerated rendering and WebAssembly-backed peer-to-peer networking via WebRTC, FlockJS provides a high-performance platform for fully in-browser interactive applications. Its declarative API abstracts low-level graphics and networking logic, allowing developers to build synchronized multiplayer games in under 100 lines of code. We present the architecture of FlockJS, including a hybrid super-peer model that reduces latency, and evaluate its performance across varying peer counts and network conditions. Results show that FlockJS sustains over 50 FPS with sub-70ms latency in hybrid topologies involving five peers. Usability feedback from student developers highlights the API's clarity and ease of integration, with requests for enhanced debugging support. By merging modern web standards into an extensible and approachable framework, FlockJS advances not only multiplayer game development but also serves as a valuable teaching tool for project-based learning in computer graphics and networking.

Keywords: Peer-to-peer networking · WebGPU · WebAssembly (WASM) · browser-native game engine · multiplayer game development · hybrid super-peer topology · real-time rendering · declarative API design

1 Introduction

The rising popularity of browser-based multiplayer games has created a demand for development tools that are not only high-performance but also accessible to independent developers and small teams. Traditional multiplayer architectures often rely on centralized servers, which introduce significant costs, latency bottlenecks, and single points of failure. At the same time, rendering frameworks such as WebGL, while widely supported, lack the low-level control needed to unlock modern GPU performance for dynamic, real-time applications.

FlockJS addresses these dual challenges by integrating **peer-to-peer networking**, **WebGPU rendering**, and **WebAssembly (WASM)** into a lightweight, modular game engine optimized for the browser. Peer-to-peer (P2P) networking enables players to communicate directly with one another without routing all

interactions through a central server, significantly reducing infrastructure overhead and latency. WebGPU, a modern graphics API currently being adopted in major browsers, allows direct access to the GPU for rendering complex visual scenes with minimal overhead. WebAssembly serves as a bridge between Rust-based networking logic and high-level JavaScript, enabling seamless integration between performance-critical components and user-facing APIs.

This paper presents the design, implementation, and evaluation of **FlockJS**, a browser-native 2D multiplayer game engine that leverages these technologies to democratize game development. FlockJS provides an easy-to-use JavaScript API for game designers while abstracting away the complexity of P2P synchronization and GPU programming. The system supports scalable multiplayer interactions through a hybrid network model using super peers and achieves high-performance rendering via batched GPU pipelines.

We describe the architectural design of FlockJS, analyze its performance under varying conditions, and report usability feedback from both developers and players. Our goal is to demonstrate that real-time, high-fidelity multiplayer games can be built and run directly in the browser without centralized infrastructure, lowering the barrier to entry for aspiring game creators.

2 Related Work

Web-based game development has advanced significantly over the past decade, driven by improvements in both client-side rendering technologies and real-time communication protocols. Early browser games relied heavily on HTML5 and WebGL for rendering, typically paired with centralized client-server architectures to manage game state. While effective in controlled environments, these setups impose scalability limits and introduce single points of failure, making them less suitable for dynamic multiplayer scenarios.

Peer-to-peer communication in browser environments is now primarily enabled by WebRTC, which facilitates real-time data exchange between clients without persistent servers. Several systems have leveraged this capability. For instance, WebTorrent utilizes WebRTC for decentralized file sharing, while Colyseus provides an abstraction layer for multiplayer state management using a hybrid model that still relies on central session coordination. OpenPix [1] offers a modular framework for building WebRTC-based multiplayer games, although it retains dependencies on server orchestration and does not address GPU integration. Dantas et al. [3] introduced a CRDT-based synchronization scheme for peer-to-peer games, effectively handling eventual consistency but not rendering performance. Rodríguez et al. [9] evaluated trade-offs in peer-driven synchronization but focused on backend-optional mobile architectures. GameBeam [6] further advanced serverless multiplayer design by combining operational transformation with peer-to-peer state exchange, improving fault tolerance and responsiveness in decentralized game sessions.

In terms of graphics, WebGL has served as the browser's default GPU interface, providing a basic abstraction layer over low-level rendering operations.

However, its limitations—such as lack of pipeline control and outdated shader models—have led to the emergence of WebGPU. As a next-generation API, WebGPU offers more explicit access to GPU capabilities, including direct control over shaders, memory, and pipelines. Chickerur et al. [2] demonstrated that WebGPU consistently outperforms WebGL across a range of devices and workloads, especially those involving complex shader logic. Fransson et al. [4] benchmarked WebAssembly-WebGPU pipelines, further validating the performance gains of GPU-native execution over traditional JavaScript-based rendering.

WebAssembly (WASM) has also emerged as a cornerstone of high-performance browser computation. It allows low-level languages such as C++ and Rust to be compiled for execution in the browser, offering near-native performance. Trautwein et al. [10] proposed a modular WASM architecture emphasizing low-overhead interoperability between host and guest languages—an approach that aligns with FlockJS, where performance-critical peer-to-peer logic is written in Rust and exposed to JavaScript through WASM bindings.

While several engines and frameworks address either peer-to-peer networking or GPU-accelerated rendering, few unify these capabilities into a cohesive model. FlockJS contributes to this emerging design space by offering an integrated, browser-native game engine that abstracts both WebGPU and WebRTC through a declarative JavaScript API. This positions it as a novel platform for building scalable, low-latency multiplayer games entirely in the browser without backend infrastructure.

3 Methods

FlockJS is a browser-native game engine that integrates peer-to-peer networking with GPU-accelerated rendering via WebGPU. It is built with modular Rust and JavaScript components linked through WebAssembly (WASM). This section outlines the system architecture, peer networking model, WebAssembly integration, and game engine design.

3.1 System Overview

FlockJS separates responsibilities between developers and the engine itself. Figure 1 shows the full sprite lifecycle, from initialization to rendering, indicating which tasks are handled by developers and which are automated by the engine.

The engine initializes a WebGPU rendering context, connects to other peers through WebRTC, and handles sprite rendering and synchronization internally.

3.2 Peer-to-Peer Networking and Super-Peer Election

FlockJS uses WebRTC connections initialized through Matchbox signaling. To minimize the cost of full mesh topologies, it optionally supports a hybrid model with a dynamically selected "super-peer" for relaying data. Figure 2 compares the two models.

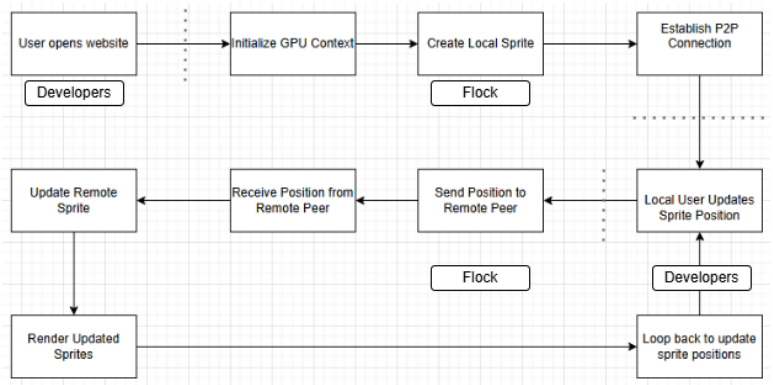

Fig. 1. Flow of sprite rendering and synchronization between peers. Boxes labeled "Flock" are automatically handled by the engine.

Super-peer selection is based on a peer quality score defined as:

$$\text{Score}(i) = \alpha \cdot \text{Upload}_i + \beta \cdot (1 - \text{RTT}_i)$$

where α and β are tunable hyperparameters, Upload_i is measured peer bandwidth, and RTT_i is round-trip time. This allows the system to dynamically choose peers with better upload capacity and latency to optimize communication.

3.3 WebAssembly Integration

FlockJS compiles its networking logic from Rust into WebAssembly (WASM) to enable high-performance peer management in the browser. Compared to traditional JavaScript, WASM offers significant gains in memory management and runtime efficiency across major browsers [9]. JavaScript loads and interacts with the compiled WASM module using bindings such as:

```
// JavaScript binding to start WebRTC peer server
wasmModule.startServer();
console.log("P2P server started");
```

This module handles ICE candidate negotiation, session description exchange, and packet transmission. WASM also buffers networking data for seamless handoff to the game logic.

3.4 Game Engine Abstractions and API Design

The engine offers a declarative API that abstracts complex WebGPU and WebRTC operations. Developers define scenes and sprite logic through a TypeScript interface. A simplified usage example:

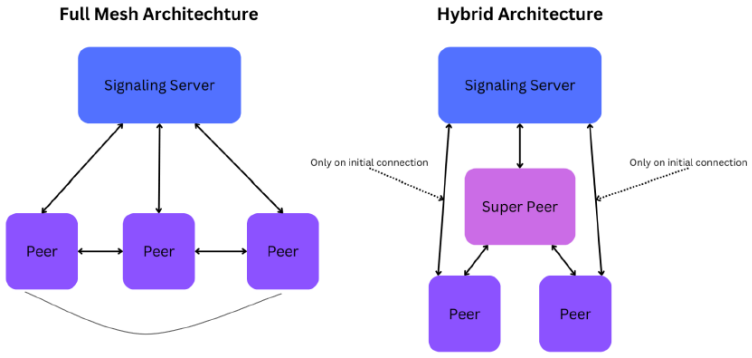

Fig. 2. Peer-to-peer network topologies supported by FlockJS: Full mesh (left) and hybrid architecture with super-peer election (right).

```
// Game scene definition in FlockJS
const gameScene: Scene = {
  id: "demo",
  preload: () => loadAssets(),
  create: (state) => {
    createSprite(state, "avatar", "user-controlled");
    wasmModule.startServer();
  },
  update: async (state) => {
    updateLocalPosition(sprite, state);
    sendLocalPosition(sprite.position);
    receiveRemotePositions(state);
    drawSprites(state); // GPU rendering handled here
  }
};
```

3.5 WebGPU Rendering Pipeline

FlockJS builds on top of the low-level WebGPU API to efficiently render 2D sprite scenes. Internally, the engine manages buffer creation, shader compilation, and frame submission. Figure 3 shows how scene objects, user inputs, shaders, and WASM modules interact in the system.

All rendering passes are optimized to reduce draw calls and memory transfers. Compute shaders can optionally be injected for tasks like animation or predictive movement.

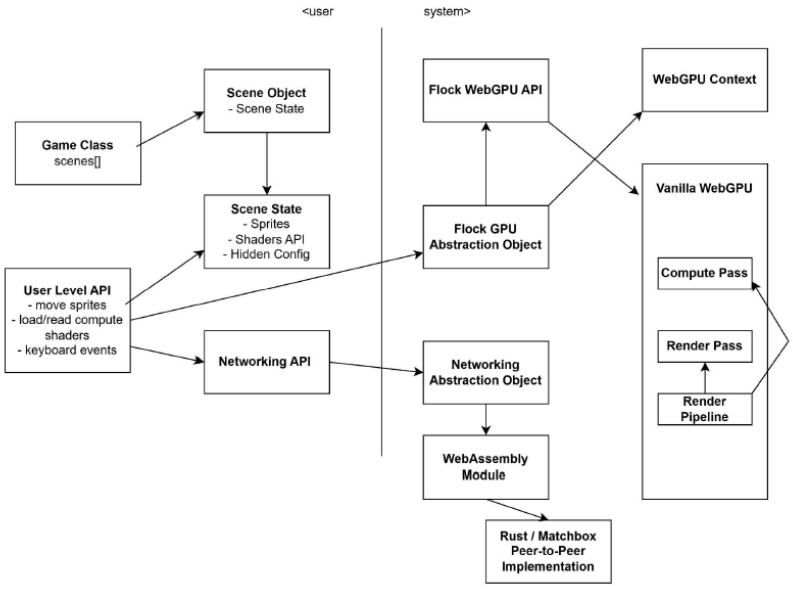

Fig. 3. Game engine architecture: left shows developer-accessible API; right shows system internals using WebGPU and WebAssembly.

3.6 Developer Workflow and Extensibility

FlockJS is designed to facilitate both rapid prototyping and modular extensibility. Developers interact with the system through a declarative JavaScript API, enabling them to register multiple scenes, define entity behaviors, and compose logic with minimal boilerplate. Advanced users can inject custom shaders directly into the WebGPU pipeline, extend core modules via WebAssembly bindings, or override default behaviors in the physics and networking subsystems.

This architecture separates low-level platform-specific complexity from user-facing code, streamlining onboarding for novice developers while preserving full control for experts. The modular design supports use cases ranging from educational demos to scalable production-grade multiplayer games. Instructors can adopt FlockJS for teaching web-based game development, while independent developers can incrementally replace components to suit their performance or gameplay requirements.

4 Results

We evaluated FlockJS across two dimensions: system performance under varying network conditions, and developer usability during real-world prototyping. The tests were conducted using Chrome 124 and Firefox 125 on Windows and

macOS devices, over home and campus Wi-Fi networks, with 1–5 browser peers simultaneously connected.

4.1 Performance Evaluation

We measured average latency and frame rate under different peer loads and simulated packet delays. Table 1 summarizes the results.

Table 1. Average round-trip latency and frame rate with increasing peer count (Full Mesh Topology)

Number of Peers	Avg Latency (ms)	Avg Frame Rate (FPS)
2	45	58
3	55	56
4	75	52
5	110	48

In the default full-mesh configuration, latency increased linearly with the number of peers due to the $O(n^2)$ connection complexity. To mitigate this, we evaluated the hybrid architecture with super-peer routing (Figure 2). Table 2 shows improved latency under the same conditions.

Table 2. Average latency using hybrid architecture with super-peer (5 peers)

Topology	Avg Latency (ms)
Full Mesh	110
Hybrid (1 Super Peer)	62

Rendering performance remained stable across network conditions. GPU draw time per frame remained under 5ms for scenes with 50+ moving sprites using compute shaders. Sample frame rendering log:

```
[GPU] Frame 51: draw_time = 4.82ms, sprites = 56
[GPU] Frame 52: draw_time = 4.77ms, sprites = 58
```

4.2 Developer Usability

To evaluate the usability of FlockJS from a developer perspective, we conducted a study with five student teams tasked with prototyping browser-based multiplayer mini-games using the engine. The teams were composed of undergraduate students with varying levels of experience in web development and game

design. Qualitative feedback was collected through structured surveys and semi-structured interviews, and responses were thematically analyzed.

Key findings are summarized as follows:

- **API Clarity:** All teams reported that the declarative API for sprites, scenes, and peer management was intuitive and required minimal boilerplate. The abstraction layers were particularly well-received by developers new to WebGPU or WebRTC.
- **Setup Experience:** Integration of WebAssembly modules was seamless from the users' perspective. Teams did not need to directly modify or understand Rust code, suggesting that the WASM boundary was well-encapsulated.
- **Debugging Challenges:** Several teams noted difficulty in debugging real-time peer state and connection stability, particularly in full mesh mode. The absence of real-time logging or visual network diagnostics emerged as a recurring theme in the feedback.
- **Documentation and Learning Resources:** While basic usage was accessible, most teams requested additional visual examples, video walkthroughs, and annotated code templates to help bridge the gap between prototype and production-level usage.

Despite these limitations, all teams successfully implemented core multiplayer features, such as synchronized sprite movement, scene transitions, and live peer discovery, in fewer than 100 lines of JavaScript. This suggests that FlockJS offers a low barrier to entry for real-time multiplayer development in the browser while maintaining extensibility for more advanced users.

4.3 Limitations Observed

Under high packet loss (simulated at 20%), peer desynchronization occurred due to the lack of reliable ordering in WebRTC's unordered data channels. We plan to incorporate optional reliability layers in future versions.

5 Discussion

The results demonstrate that FlockJS can enable real-time, scalable peer-to-peer multiplayer experiences with minimal backend infrastructure. By combining a hybrid networking model with GPU abstraction via WebGPU, the system achieves a balance between rendering performance and developer accessibility. This section reflects on the architectural trade-offs, design implications, and potential directions for future development.

5.1 Architectural Trade-offs

Our evaluation of full mesh and hybrid peer-to-peer models underscores a central scalability trade-off. Full mesh architectures ensure direct communication

between all peers but introduce $O(n^2)$ connection complexity, which results in elevated bandwidth usage and latency as the number of users increases. In contrast, the hybrid model elects a single super-peer to relay data, improving scalability while introducing the risk of centralized bottlenecks. This trade-off must be managed dynamically based on peer performance metrics.

Compared to conventional client-server topologies, FlockJS's peer-to-peer architecture reduces server load and can improve responsiveness in small to medium-scale deployments. Nevertheless, issues such as peer churn, NAT traversal, and firewall restrictions persist. While FlockJS incorporates fallback signaling and lightweight heuristics for routing, integration with robust solutions such as TURN servers is planned for future iterations.

5.2 Design Insights

A core insight from the development of FlockJS is the advantage of cleanly separating game logic from low-level rendering and networking primitives. By offering a high-level, declarative API and delegating computationally intensive tasks to WebAssembly modules, FlockJS significantly lowers the technical barrier for developers—particularly students and early-stage prototypers. This modularity not only simplifies onboarding but also fosters rapid experimentation and iterative design. The effectiveness of declarative paradigms in interactive system development has been well-documented [11]; FlockJS extends these benefits into the domain of browser-native, multiplayer game engines.

The engine is also tailored for educational contexts. Recent research on the integration of generative AI tools in CS education has emphasized the value of modular, visual, and interactive frameworks in enhancing student understanding [8]. FlockJS complements these trends by enabling hands-on exploration of real-time systems, distributed networking, and graphics programming—all within a unified web environment. Moreover, its project-oriented architecture aligns with pedagogical strategies that promote authentic, portfolio-driven learning, similar to efforts to embed HCI datasets into undergraduate machine learning courses [7].

Our experience with WebGPU further illustrates this dual emphasis on performance and accessibility. WebGPU's low-level control over GPU pipelines enables considerable performance gains in sprite batching and shader execution. However, its limited browser support and steep learning curve present practical challenges. FlockJS addresses these by abstracting away complexity through internally layered APIs, balancing the power of WebGPU with the ease of high-level scripting.

5.3 Limitations

Despite its strengths, FlockJS exhibits several limitations:

– **Packet Reliability:** WebRTC's data channels lack built-in reliability guarantees, leading to occasional synchronization errors during packet loss.

- **Debugging Overhead:** Real-time debugging of peer state and network latency remains cumbersome, even with extensive logging. Prior work [5] highlights the importance of visual, event-driven debugging tools to reduce cognitive load and improve developer comprehension in multiplayer systems—an area that FlockJS currently lacks.
- **Mobile Compatibility:** The current implementation is optimized for desktop browsers and has not yet been tuned for mobile GPUs or touch-based input handling.

5.4 Future Work

5.5 Future Work

Future versions of FlockJS may integrate improvements along the following lines:

- **Reliable Messaging Layers:** Incorporating optional reliability protocols (e.g., FEC, selective retransmission) over WebRTC could bolster network robustness.
- **Visual Debugging Tools:** An interactive in-browser panel for displaying peer connections, network metrics, and real-time sprite states would aid in troubleshooting.
- **Mobile Optimization:** Adapting input handlers and memory usage patterns to support mobile browsers will expand accessibility.
- **Modular Physics and CRDTs:** Inspired by recent work on composable multiplayer systems [3], we plan to explore CRDT-based conflict resolution and modular physics engines to support extensible real-time applications.
- **Scene Partitioning:** Supporting region-based loading and spatial partitioning will improve scalability for larger maps and multi-region play.
- **Educational Tool Extensions:** Following the design philosophy of BrainActivity1 [12], future iterations of FlockJS may offer educational dashboards and simplified scripting environments to further support classroom use and student-led experiments in real-time multiplayer design.

These extensions aim to transition FlockJS from a prototyping and teaching framework into a general-purpose game engine suitable for indie and commercial multiplayer applications.

6 Conclusion

FlockJS demonstrates the feasibility of browser-native, peer-to-peer multiplayer game development using modern web technologies such as WebGPU and WebAssembly. By combining declarative APIs with a modular architecture, the engine lowers the barrier to entry for developers while maintaining strong performance across a range of network conditions.

Our results show that hybrid peer-to-peer topologies can significantly reduce latency compared to full mesh networking, and that WebGPU-based rendering

maintains consistent frame rates even with dozens of sprites in motion. Developer feedback confirms the usability of FlockJS in educational and prototyping contexts, with positive responses to its API design and low setup overhead.

Looking ahead, enhancements such as reliability layers, improved debugging support, and mobile optimization will broaden the applicability of the engine. FlockJS contributes to the democratization of multiplayer game development by making scalable, high-performance networking and rendering accessible directly within the browser.

References

1. Borges, R.C., Malheiros, M.d.G., Billa, C.Z., Pias, M.R., Bicho, A.d.L.: An open-source framework using webrtc for online multiplayer gaming. In: Proceedings of the 22nd Brazilian Symposium on Games and Digital Entertainment. pp. 143–150 (2023)
2. Chickerur, S., Balannavar, S., Hongekar, P., Prerna, A., Jituri, S.: Webgl vs. webgpu: A performance analysis for web 3.0. Procedia Computer Science **233**, 919–928 (2024)
3. Dantas, A., Baquero, C.: Crdt-based game state synchronization in peer-to-peer vr. In: Proceedings of the 12th Workshop on Principles and Practice of Consistency for Distributed Data. pp. 45–55 (2025)
4. Fransson, E., Hermansson, J.: Performance comparison of webgpu and webgl in the godot game engine (2023)
5. Mehanna, N., Rudametkin, W.: Caught in the game: On the history and evolution of web browser gaming. In: Companion Proceedings of the ACM Web Conference 2023. pp. 601–609 (2023)
6. Onat, C.: GameBeam: A Decentralized Framework for P2P Multiplayer Game Streaming. Ph.D. thesis, Worcester Polytechnic Institute (2025)
7. Qu, X., Key, M., Luo, E., Qiu, C.: Integrating hci datasets in project-based machine learning courses: a college-level review and case study. In: International Conference on Human-Computer Interaction. pp. 124–143. Springer (2024)
8. Qu, X., Sherwood, J., Liu, P., Aleisa, N.: Generative ai tools in higher education: A meta-analysis of cognitive impact. In: Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems. pp. 1–9 (2025)
9. Rodríguez Baquero, D.: Analysis of webrtc signaling (2021)
10. Trautwein, D., Raman, A., Tyson, G., Castro, I., Scott, W., Schubotz, M., Gipp, B., Psaras, Y.: Design and evaluation of ipfs: a storage layer for the decentralized web. In: Proceedings of the ACM SIGCOMM 2022 Conference. pp. 739–752 (2022)
11. White, W., Sowell, B., Gehrke, J., Demers, A.: Declarative processing for computer games. In: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games. pp. 23–30 (2008)
12. Zhou, Z., Dou, G., Qu, X.: Brainactivity1: A framework of eeg data collection and machine learning analysis for college students. In: International Conference on Human-Computer Interaction. pp. 119–127. Springer (2022)