

A layered browser-native architecture for real-time multi-window HUD simulation

A Case Study of the Yak-130 HUD Display

Tran Cong Minh, Do Van Quynh, Nguyen Hong Viet, Nguyen Anh, Pham Ba Luan

Faculty of Radio-Electronics

Air force Officer's College

Khanh Hoa, Viet Nam

Email: minh.tc160593@gmail.com

Abstract— This paper presents a layered browser-native architecture for real-time aircraft HUD simulation. The architecture introduces modular separation of simulation components together with a lightweight state-distribution mechanism that enables low-latency synchronization across multiple browser windows. A unified rendering pipeline integrates two-dimensional HUD symbology with a conformal three-dimensional visualization while preserving modularity between simulation and visualization components. The proposed architecture is validated using a Yak-130 HUD simulator and evaluated for functional correctness, real-time performance, and scalability. Experimental results show that the architecture supports efficient browser-based HUD simulation without requiring dedicated hardware or a compilation toolchain.

Keyword— *HUD simulation; aircraft HUD; layered software architecture; real-time; browser-based simulation; loose coupling; producer-consumer.*

I. INTRODUCTION

HUD simulators play an increasingly important role in flight training: they allow pilots and trainees to practice reading and interpreting the display, to master the avionics, and to rehearse a wide range of flight situations without relying on real aircraft, which are costly and difficult to access. This raises a central question: how can one build a real-time HUD simulator that is low-cost, easy to deploy, and easy to extend?

Existing systems — from dedicated desktop applications and game engines such as Unity and Unreal Engine, to FlightGear, DCS, and models built in MATLAB — achieve high fidelity but share common limitations: they require installation, depend on the platform and operating system, need a compile-package workflow, are hard to extend, and do not easily synchronize multiple displays in a lightweight manner. This is precisely the gap that needs to be filled.

Conversely, modern web browsers already provide the capabilities needed to build a simulator: Canvas 2D for drawing symbology, Three.js/WebGL for rendering

3D scenes, the Gamepad API for reading a joystick, and BroadcastChannel for exchanging data among windows. However, platform capability is not the same as a general architecture for HUD simulation: most web-based work stops at visualization and does not address the software-architecture problem of generating and distributing HUD state in real time to multiple views. This is the research gap addressed in this paper.

The main contributions of this paper are as follows:

- **Layered architecture.** We propose a layered architecture for a real-time browser-based HUD simulator that cleanly separates the control, flight-model, state, presentation, and distribution concerns, thereby improving extensibility and reuse.
- **Synchronization mechanism.** We propose a producer-consumer state-distribution mechanism based on immutable snapshots with latest-state-wins semantics, which synchronizes multiple display windows at low latency without locks or complex queues.
- **Real-time pipeline.** We propose a unified display pipeline that combines the flight model, HUD-state generation, Canvas 2D rendering, and a conformal Three.js 3D scene entirely within the browser.
- **Validation.** We validate the architecture through the Yak-130 HUD display, evaluating functional correctness, real-time performance, and multi-window scalability.

It should be emphasized that the contribution lies neither in the Yak-130 nor in “writing a HUD in JavaScript,” but in a software architecture that is applicable to many different HUD systems; the Yak-130 merely demonstrates that the architecture works effectively in a realistic case. The remainder of the paper is organized as follows. Section 2 reviews related work; Section 3 presents the proposed architecture; Section 4 describes the implementation; Section 5 reports the experimental evaluation; Section 6 discusses the results; and Section 7 concludes.

II. RELATED WORK

A. HUD and cockpit simulators

Traditional flight and cockpit simulators are typically built as desktop applications or on top of game engines. Dedicated applications and systems such as FlightGear and DCS achieve high fidelity and detailed flight dynamics [2], [3], [17]; game-engine-based systems (Unity, Unreal) excel at graphics and interactivity. Nevertheless, both are tightly bound to their platforms, require installation and a compile-package workflow, and typically rely on networking infrastructure or platform-specific mechanisms to synchronize multiple displays.

B. Web-based simulation

With the maturity of WebGL [16], Canvas 2D, and libraries such as Three.js [15], many 3D-visualization systems, digital twins, and browser-based flight simulators have emerged. These works demonstrate that the browser is capable of real-time rendering and interaction, but most focus on rendering and visualization for a single view and do not address the software-architecture problem of generating and distributing HUD state in real time to multiple independent windows.

C. Research gap

Table 1 summarizes the capabilities of the different categories of work against four core criteria: whether they run in the browser, support multiple windows, adopt a decoupled architecture, and operate in real time.

Existing work focuses mainly on visualization or single-view desktop simulation; approaches that support multiple windows and decoupling well (such as distributed simulation) require heavyweight infrastructure.

Table 1: Survey of related-work categories against four core criteria.

Category of work	Runs in browser	Multi-window	Decoupled architecture	Real-time
Desktop cockpit simulators (FlightGear, DCS...)	No	Partial	Partial	Yes
Game-engine simulators	Partial	No	Partial	Yes
Web visualization / digital twins (Three.js, Cesium)	Yes	No	No	Partial
Distributed simulation (HLA/DIS)	No	Yes	Yes	Yes (heavyweight)
Proposed architecture	Yes	Yes	Yes	Yes

No general, lightweight, multi-window layered architecture exists for real-time HUD simulation that runs directly in the browser. This paper fills that gap.

III. PROPOSED ARCHITECTURE

Fig.1 gives an overview of the architecture: the entire system runs inside a commodity browser, with a HOST window that contains the simulation core (control-physic-state) and the state-distribution mechanism, feeding synchronized data to multiple display windows. The system requires only commodity input devices, with no installation and no dedicated hardware.

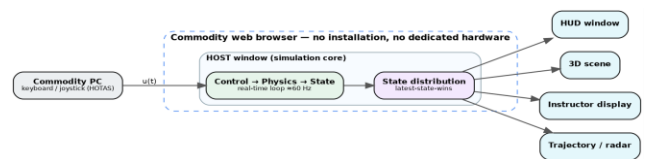


Fig. 1: Architecture overview: one browser, one HOST window as the single source of truth, synchronizing multiple display windows.

A. Design requirements

Before describing the structure, it is necessary to state precisely what the architecture must achieve. Table 2 lists the requirements and how the architecture meets them.

Table 2: Design requirements and how the architecture meets them.

Requirement	Description	How the architecture meets it
Real-time (≈ 60 Hz)	Smooth updates synchronized with the display refresh rate	requestAnimationFrame loop with time-step clamping
Low latency	State reaches display windows quickly, with no backlog	Latest-state-wins; no queue
Portability	Runs on many machines/OSes without modification	Uses only standard web APIs
No installation / compilation	Open the files and run	Pure web application, no build tooling
Modularity	One concern per module	Layering with explicit interfaces
Extensibility and reuse	Convenient replacement or addition of components	Loose coupling via the snapshot contract
Multi-window synchronization	Consistent multiple views	Single source of truth plus a distribution channel

B. Layered architecture

The system comprises six layers with one-way, top-down dependencies (Fig. 2): input, flight dynamics (physics), state, presentation, distribution, and display. Each layer exposes only a narrow interface to the next and does not access the internals of any other layer.

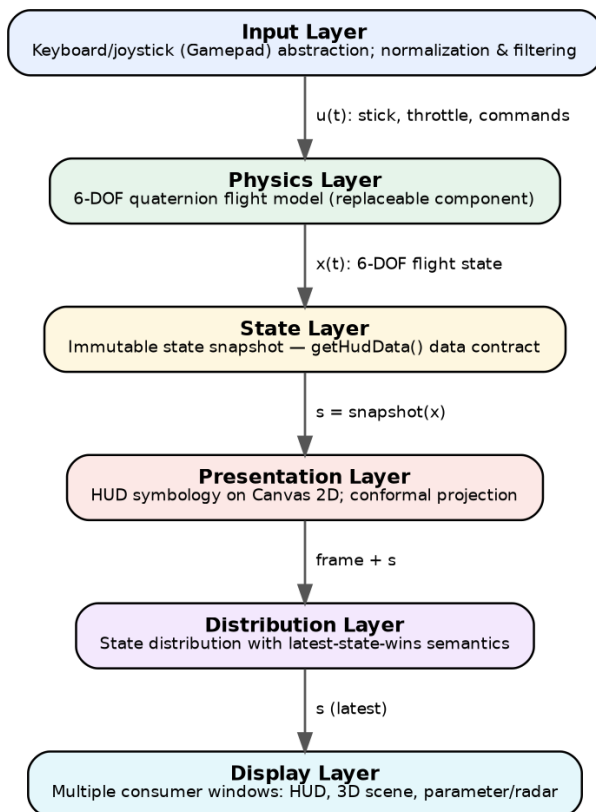


Fig 2: Layered architecture with one-way dependencies; layers communicate through a data contract.

The state layer acts as the single intermediate interface of the architecture: it defines a common data contract that fully decouples the data-producing side (physics) from the data-consuming side (presentation, display), so that the two sides can evolve independently as long as they honor the shared contract. Table 3 summarizes the responsibility and interface of each layer.

Table 3: Responsibilities and interfaces of the six layers.

Layer	Main responsibility	Input	Output
Input	Abstracts and normalizes keyboard/joystick signals	Device events	Control vector $u(t)$
Physics	Integrates 6-DOF flight dynamics (replaceable component)	$u(t)$	Flight state $x(t)$
State	Builds the immutable snapshot — getHudData() contract	$x(t)$	Snapshot s
Presentation	Generates HUD symbology (Canvas 2D); conformal projection	s	HUD frame
Distribution	Publishes the snapshot with latest-state-wins semantics	s	Snapshot stream
Display	Multiple consumer windows render in their own loops	s (latest)	On-screen image

C. Data flow

Fig. 3 shows the data flow under the producer–consumer model. Within the HOST window (the producer), data flows sequentially: keyboard/joystick → flight model → HUD state (an immutable snapshot) →

presentation → distribution. From the distribution point, the state is pushed to the consumer windows (HUD, 3D scene, instructor/trajectory). The state contract s packages all the information needed to draw the HUD at a given instant — pitch, roll, yaw, flight-path angle, angle of attack, airspeed, altitude, load factor, localizer/glideslope deviation, mode labels, and runway geometry — and is the stable interface between the two sides.

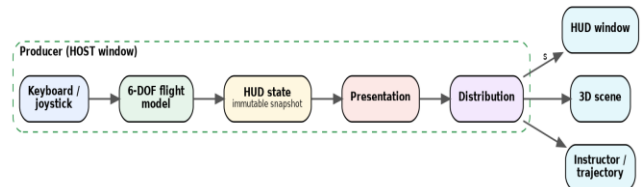


Fig. 3: Producer–consumer data flow: from control input to the display windows.

D. Synchronization strategy

The synchronization strategy is one of the main contributions and is characterized by four principles that are independent of the implementation tools:

- **Snapshot distribution:** only an immutable state structure is distributed; no reference to the mutating state is shared.
- **Latest-state-wins:** a new message overwrites the previous one and no queue is maintained, so latency does not accumulate.
- **Stateless consumer:** each window renders only the latest state it receives and keeps no history.
- **Asynchronous rendering:** each window has its own render loop, decoupling the rates of the producer and the consumers.

Fig. 4 illustrates the mechanism: the HOST window is the single source of truth and publishes the state through a channel to the consumer windows. The significance of the latest-state-wins principle becomes clear when compared with a queue-based mechanism (Fig. 5): with a queue, whenever a consumer window is temporarily slower than the producer, messages accumulate and latency grows without bound over time; with latest-state-wins, in contrast, latency is bounded to about one frame and is independent of the number of windows:

$$L_{consumer} \leq T_{frame} + \varepsilon \quad (1)$$

independent of the number of windows N .

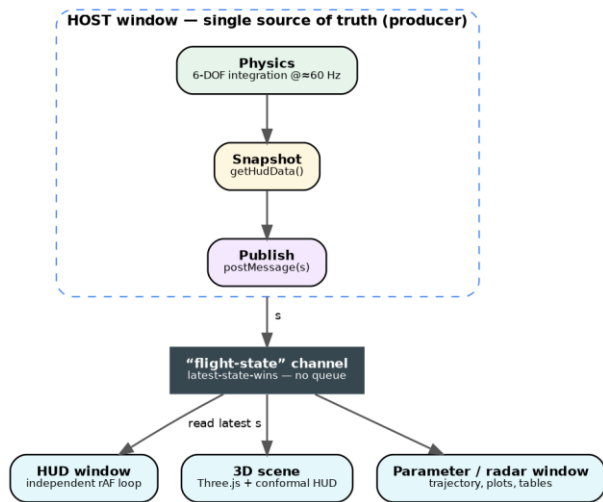


Fig. 4: Producer-consumer synchronization mechanism with a single source of truth.

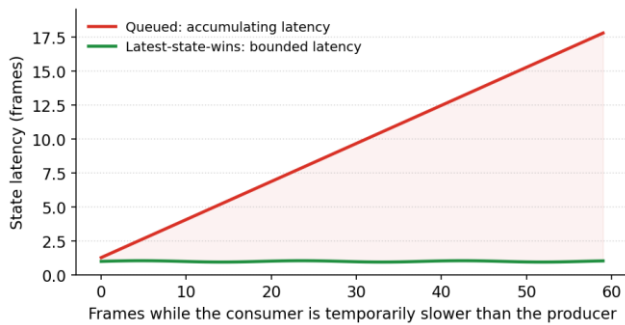


Fig. 5: Conceptual illustration: a queue accumulates latency, whereas latest-state-wins keeps latency bounded.

It is important to distinguish the design pattern from the implementation tool: the producer-consumer model based on immutable snapshots is the architectural idea, whereas BroadcastChannel is merely one implementation option that can be replaced by a SharedWorker or a WebSocket without changing the architecture.

E. Rendering pipeline

The rendering pipeline (Fig. 6) takes the state contract as its sole input and splits it into two conformal branches: the Canvas 2D branch draws the symbology (pitch ladder, flight-path vector, speed/altitude scales, heading), while the Three.js branch renders the scene-locked 3D elements (horizon, runway, terrain); the two branches are composited into the final HUD frame. For a stand-alone HUD, the system uses a pinhole camera model to project a world point P onto the screen:

$$f = \frac{W}{2 \cdot \tan(FOV/2)}; x_s = \frac{rel-right}{depth} \cdot f; y_s = -\frac{rel-up}{depth} \cdot f \quad (2)$$

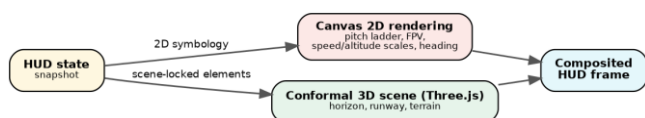


Fig. 6: Rendering pipeline: from the HUD state to the Canvas 2D and conformal 3D branches.

where $rel = P - camera_position$ and $depth = rel-forward$; when the HUD is overlaid on the 3D scene, points are projected through the Three.js perspective camera itself to align exactly with the terrain. The entire rendering stage resides in the presentation layer and depends only on the state contract, and is therefore independent of how the state is produced.

F. Modifiability and extensibility

Because the layers communicate only through the snapshot contract, every valid change is localized to the responsible layer (Table 4). This is concrete evidence of loose coupling: the aircraft, the symbology set, an additional display, or the transport can each be changed without propagating to the other layers.

Table 4: Impact locality of typical changes — evidence of loose coupling.

Change scenario	Layer to modify	Unchanged layers	Remark
Change aircraft (different flight model/parameters)	Physics	Input, State, Presentation, Distribution, Display	Snapshot contract unchanged
Change HUD symbology (different convention)	Presentation	Physics, State, Distribution, Display	Different rendering; source data unchanged
Add a new display (e.g. engine page)	Display (+1 consumer)	Physics, State, Presentation, Distribution	Only add one window that subscribes to the channel
Change transport (BroadcastChannel → WebSocket)	Distribution	Physics, State, Presentation, Display	Different implementation; architecture unchanged

IV. IMPLEMENTATION

The architecture is realized as the Yak-130 HUD simulator, built as a pure web application: the HTML files are opened directly, with no compilation or packaging tooling. The four main components corresponding to the layers are described below.

A. Flight model

The flight model is a six-degree-of-freedom rigid body that represents attitude with a unit quaternion to avoid gimbal lock, computes aerodynamic forces and moments by a coefficient build-up method, and uses an exponential atmosphere model. The equations are integrated with a semi-implicit Euler scheme, one step per frame, and the state is clamped to physical limits after each step; the attitude kinematics are updated as:

$$q \leftarrow \text{normalize} \left(q + \frac{1}{2} \cdot (q \otimes \omega) \cdot \Delta t \right), \omega = (0, p, q, r) \quad (3)$$

where (p, q, r) are the body-frame angular rates. The model is configured as an engineering-level aircraft flight model to provide physically plausible responses for HUD visualization and training scenarios. It reproduces the expected qualitative relationships

among attitude, angular motion, airspeed, altitude, and flight-path behavior rather than targeting high-fidelity aerodynamic prediction. Because the flight model is isolated behind the snapshot interface, it can be replaced or reconfigured without affecting the upper layers.

B. HUD symbol generator

The presentation layer uses Canvas 2D to draw the symbology in the characteristic green color: the pitch ladder, heading indicator, speed and altitude scales, the flight-path vector (FPV), and the mode labels. The Cyrillic mode labels are preserved as on the original device (ВЗЛ - takeoff, ПОС - landing, HAB - navigation, ИЛС - HUD). The symbology is generated directly from the state contract, ensuring consistency across all windows.

C. Conformal 3D scene

The 3D scene is rendered with Three.js using a perspective camera matched to the HUD field of view, and comprises the horizon, the runway strip, and reference terrain. The scene-locked elements are projected through the 3D camera to align with the real world, while the non-conformal symbology remains the responsibility of Canvas 2D; the two layers are composited to form the complete HUD view.

D. Multi-window deployment

Thanks to the state-distribution mechanism, a single browser can open several windows simultaneously - the HUD, the instructor display, the trajectory window, and the 3D scene - all synchronized to the HOST window. This capability is hard to achieve in a lightweight manner on desktop platforms or game engines. Fig. 7 illustrates the multi-window deployment layout (a screenshot is to be inserted).

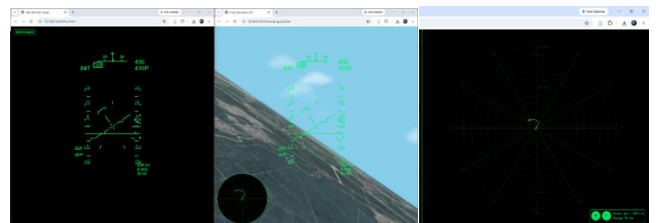


Fig. 7: Synchronized multi-window deployment in a single browser

V. EXPERIMENTAL EVALUATION

A. Experimental setup

Table 5 summarizes the hardware and software configuration of the experimental platform used in this study.

Component	Configuration (fill with actual values)
CPU	Intel Core i5-12450HX
Memory (RAM)	16 GB DDR5
GPU	NVIDIA RTX 3050 Laptop GPU 6 GB
Operating system	Windows 11 Pro
Browser (and version)	Google Chrome 150
Display / resolution / refresh rate	1920×1080 @ 144 Hz
Joystick	Logitech Extreme 3D Pro

B. Functional validation

Functional correctness is evaluated by verifying that the proposed architecture correctly generates, propagates, and renders the HUD state throughout the complete browser-native simulation pipeline. Fig. 8 presents a representative output of the simulator, illustrating the conformal alignment between the generated HUD overlay and the corresponding three-dimensional scene. The evaluation focuses on the correctness of HUD-state generation, conformal registration, consistent rendering, and dynamic display behavior across the synchronized visualization pipeline.

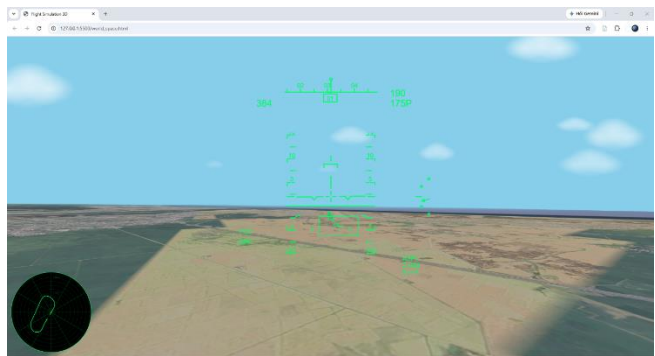


Fig. 8: Conformal 3D scene with HUD overlay.

C. Scalability evaluation

To evaluate the scalability of the proposed architecture, the number of consumer windows was gradually increased from one to five. For each configuration, snapshots were time-stamped at the HOST, and the state latency at each consumer window was calculated as the difference between the snapshot timestamp and the reception time. In addition, the frame rate, CPU utilization, and memory consumption were recorded. The experimental results are presented in Table 7.

Table 5: Experimental platform.

Table 7: Scalability results by window count: frame rate, latency, and resources.

Consumer windows	HOST frame rate (FPS)	Average consumer frame rate (FPS)	Average state latency (ms)	RAM (MB)	CPU (%)
1	144.1	144.1	4.8	150	10.8
2	143.8	143.8	5.0	204	12.0
3	143.4	143.1	5.0	260	15.6
4	143.3	143.3	5.2	320	18.0

As the number of consumer windows increases from one to four, the proposed architecture maintains essentially constant real-time performance. The HOST frame rate remains close to the display refresh rate (approximately 144 FPS), while the average frame rate of the consumer windows closely follows that of the HOST, indicating that asynchronous rendering introduces negligible performance degradation. More importantly, the average state latency increases only slightly, from 4.8 ms to 5.2 ms, remaining well below the duration of a single display frame. This behavior is consistent with the latest-state-wins synchronization strategy, which prevents latency accumulation by ensuring that each consumer always renders the most recent snapshot rather than processing queued updates.

Resource consumption increases approximately linearly with the number of consumer windows, as expected because each additional window maintains an independent rendering context. Memory usage rises from 150 MB to 320 MB and CPU utilization increases from 10.8% to 18.0%, while frame rate and latency remain essentially unchanged. These results indicate that the additional computational cost is primarily associated with rendering rather than state distribution. Overall, the experiments demonstrate that the proposed layered architecture scales efficiently with the number of synchronized browser windows, providing predictable resource growth without compromising real-time responsiveness or synchronization accuracy.

D. Scalability with heterogeneous views

Beyond increasing the number of windows, several heterogeneous view types — the HUD, the 3D scene, the trajectory, the instructor display, and the parameter/radar window — are enabled simultaneously to test the architecture under a mixed load. Because the consumer windows are stateless and render asynchronously, adding a view type merely adds one consumer that subscribes to the channel and does not affect the single source of truth.

E. Comparison

Table 8 directly compares the proposed approach with a desktop application and a game engine along important features.

Table 8: Feature comparison among a desktop application, a game engine, and the proposed architecture.

Feature	Desktop application	Game engine (Unity)	Proposed architecture
Runs in browser, no installation	No	Partial	Yes
No compilation / packaging	No	No	Yes
Lightweight multi-window sync	Partial	No	Yes
Latest-state-wins distribution	No	No	Yes
Layered architecture, localized extension	Partial	Partial	Yes
Platform / hardware independence	Partial	Partial	Yes
Deployment / maintenance cost	High	High	Low

VI. DISCUSSION

Two design choices carry the contribution. First, the layering makes the cost of change predictable: because the layers meet only at the snapshot contract, retargeting the simulator to another aircraft or symbology set is confined to a single layer (Section 3.6) rather than spread across the codebase, which makes the system easy to reuse and extend. Second, the latest-state-wins distribution decouples the consumers' rates from the producer, so that, by construction, adding a window affects resource use but not the inter-window state latency; the experiments of Section 5.3 are designed to test precisely this property. Because it runs entirely in the browser, the simulator also gains portability and zero-installation deployment at no additional engineering cost.

In terms of limitations, several points must be acknowledged frankly: JavaScript and the browser do not guarantee hard real-time behavior, and the requestAnimationFrame rate depends on the browser scheduler and may exhibit jitter; Canvas 2D rendering has a performance ceiling at very high symbology densities; the system is not yet connected to real hardware and does not yet support virtual or augmented reality; and the flight model is an engineering-level model with estimated parameters, ensuring correct trends rather than absolute aerodynamic accuracy. The system is therefore suitable for training in reading and interpreting the display, but does not replace certified embedded systems in the cockpit.

Future work includes: moving the presentation layer to WebGPU for higher performance; synchronizing multiple machines (networked/multi-user) over WebSocket; integrating a motion platform; connecting HUD hardware in a Hardware-in-the-Loop configuration compatible with the ARINC-429/MIL-STD-1553

protocols; and supporting augmented-reality (AR) headsets.

VII. CONCLUSIONS

This paper has proposed a layered, browser-native architecture for real-time multi-window HUD simulation, with three contributions: a loosely coupled six-layer decomposition; a producer–consumer state-distribution mechanism with latest-state-wins semantics; and a unified rendering pipeline that combines Canvas 2D with a conformal 3D scene. The architecture has been validated through the Yak-130 HUD display and evaluated for functional correctness, real-time performance, and scalability. The architecture generalizes beyond this case: any trainer whose display state can be expressed as a snapshot can reuse the same layering and distribution mechanism, changing only the flight-model and presentation layers. Directions for future work were outlined in Section 6.

REFERENCES

- [1] U.S. Department of Defense, MIL-STD-1787D, Aircraft Display Symbolology, 2018.
- [2] D. Allerton, Principles of Flight Simulation. Chichester: John Wiley & Sons, 2009.
- [3] B. L. Stevens, F. L. Lewis, and E. N. Johnson, Aircraft Control and Simulation, 3rd ed. Hoboken, NJ: Wiley, 2015.
- [4] B. Etkin and L. D. Reid, Dynamics of Flight: Stability and Control, 3rd ed. New York: Wiley, 1996.
- [5] M. V. Cook, Flight Dynamics Principles, 3rd ed. Oxford: Butterworth-Heinemann, 2013.
- [6] J. Diebel, "Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors," Stanford University, Tech. Rep., 2006.
- [7] R. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision, 2nd ed. Cambridge: Cambridge University Press, 2004.
- [8] ISO 2533:1975, Standard Atmosphere. Geneva: International Organization for Standardization, 1975.
- [9] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed. Boston: Addison-Wesley, 2012.
- [10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Chichester: Wiley, 1996.
- [11] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pp. 40–52, 1992.
- [12] IEEE Std 1516-2010, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules. IEEE, 2010.
- [13] IEEE Std 1278.1-2012, IEEE Standard for Distributed Interactive Simulation (DIS) – Application Protocols. IEEE, 2012.
- [14] Mozilla Developer Network, "Web APIs: BroadcastChannel, requestAnimationFrame, Gamepad API, Canvas 2D." [Online]. Available: <https://developer.mozilla.org/>
- [15] Three.js, "Three.js Documentation." [Online]. Available: <https://threejs.org/docs/>
- [16] Khronos Group, "WebGL Specification." [Online]. Available: <https://www.khronos.org/webgl/>
- [17] FlightGear Flight Simulator (open-source project). [Online]. Available: <https://www.flightgear.org/>
- [18] Airforce Technology, "Yak-130 – Advanced Jet Trainer." [Online]. Available: https://www.airforce-technology.com/projects/yak_130/