Web-Native Neuroviz: Three.js + WebSockets for Live Brain Streams

Benjamin J. Gilbert
Spectrcyde
Email: bgilbert2@com.edu

***Abstract*—We present a web-native neural visualization system combining Three.js WebGL rendering with robust WebSocket streaming for real-time voxel field display. Our client sustains 60 FPS with median latency $\sim$20 ms and p99 $<$50 ms across 16^3–64^3 voxel densities, with JSONL telemetry and auto-figures for reproducible analysis. The React/TypeScript client implements exponential backoff reconnection, live performance monitoring (p50/p99 latency tracking), and comprehensive metrics export via JSONL format. Performance analysis demonstrates scalable bandwidth utilization and consistent frame timing with stutter rates below 2.5%. The complete implementation includes automated figure generation, statistical analysis tools, and a synthetic data server for development and testing.**

***Index Terms*—Three.js, WebSockets, Neural Visualization, Real-time Rendering, WebGL, Performance Analysis**

I. Introduction

Real-time visualization of neural activity data presents unique challenges in web environments, requiring both high-performance 3D rendering and reliable network streaming. Traditional approaches rely on native applications or server-side rendering, limiting accessibility and deployment flexibility. We address these limitations through a web-native architecture combining Three.js WebGL capabilities with robust WebSocket communication.

Building on advances in RF-based neural sensing and neuromodulation, our system targets live brain imaging scenarios where voxel field data must be streamed and rendered at interactive frame rates. This extends visualization capabilities for RF-derived neural data streams, supporting real-time monitoring and control applications. Key requirements include: (1) 60 FPS rendering performance, (2) sub-20ms network latency tolerance, (3) graceful handling of network interruptions, and (4) comprehensive performance monitoring for production deployment.

The contribution of this work is a complete, production-ready implementation demonstrating that modern web technologies can meet demanding real-time visualization requirements without sacrificing performance or reliability.

II. System Architecture

A. Client-Side Rendering

The visualization client leverages Three.js for WebGL-accelerated point cloud rendering. Voxel data is represented as sparse point clouds, with occupancy thresholding applied client-side to reduce vertex count. We render sparse voxel clouds as a single `THREE.Points` with a preallocated `Float32BufferAttribute`. Voxel occupancy $\geq \theta$ (default $\theta = 0.6$) yields a compact index list; updates change only the draw range to avoid memory reallocation during frame updates.

The rendering pipeline implements frustum culling (90° FOV exclusion) and two LOD tiers: full points for $\leq 32^3$ voxels, 50% decimation above 32^3 to stabilize frame time. Camera controls provide standard orbit navigation with smooth interpolation.

B. WebSocket Communication

Network communication employs WebSocket connections with exponential backoff reconnection logic. Clients accept (1) JSON frames (`{"dims":[X,Y,Z],"t":...,"values_b64":base64f32}`) and (2) a binary format with a 12-byte header (`uint32` X,Y,Z) followed by raw `Float32` payload, reducing CPU overhead by 30% vs. JSON.

Connection resilience uses jittered exponential backoff for reconnect (base 250 ms, cap 5 s, $\pm$50 ms jitter) with automatic reconnection on network failures.

C. Performance Telemetry

For each received frame we log: receive time t_{recv}, draw time t_{draw}, voxel count n_{vox}, payload bytes b, and instantaneous FPS. We compute end-to-end latency $\Delta = t_{\text{draw}} - t_{\text{recv}}$, maintain rolling percentiles (p50/p99) over the last 5000 frames, and export one JSONL line per frame. The plotting script expects keys: `ts_recv`, `ts_draw`, `frame_ms`, `latency_ms`, `voxels`, `payload_bytes`.

III. Experiments and Results

A. Experimental Setup

We sweep voxel sizes $16^3 \rightarrow 64^3$ using the synthetic WebSocket server, testing both JSON and binary modes. We log 30,000 frames per condition and generate figures via `scripts/gen_neuroviz_figs.py` from JSONL telemetry data.

B. Performance Analysis

Figure 1 shows comprehensive performance analysis from live traces. The latency histogram (left) demonstrates sub-20ms p50 performance with clear p50/p99 markers. The FPS vs. voxel count plot (middle) shows median FPS trends

across densities, while bandwidth analysis (right) reveals linear scaling relationships.

A representative live run achieved p50 = 19.8 ms, p99 = 47.8 ms, median FPS = 60.1 with stutter ratio = 2.1% for frames exceeding 25 ms. Binary WebSocket format reduces CPU overhead and stabilizes p99 latency compared to JSON encoding.

Table I summarizes key distributional metrics from live stream traces, demonstrating consistent performance across varying network conditions and voxel densities.

C. Bandwidth Utilization

The sparse voxel representation provides significant compression for typical neural data patterns, with 20-30% occupancy rates resulting in linear bandwidth scaling. Binary format achieves 30% reduction in network overhead compared to Base64 JSON encoding.

IV. Web-Native Neuroviz: Three.js + WebSockets

We present a web-native viewer that renders voxel fields at 60 fps via Three.js while ingesting live frames over a robust, auto-reconnecting WebSocket (WS). The client emits a JSONL metrics stream (latency, frame time, voxel count, payload size) for offline analysis.

a) Client.: The React/TypeScript component `web/NeuralVisualization.tsx` maintains a capped point cloud (sparse occupancy above a threshold) using a single `THREE.Points` buffer with draw-range updates. The WS layer applies exponential backoff with jitter on disconnect and measures end-to-end latency ($t_{\text{draw}} - t_{\text{recv}}$). A sliding window tracks p50/p99 latency and live FPS.

b) Figures.: We parse the JSONL log with `scripts/gen_neuroviz_figs.py` to produce: (**a**) latency histogram with p50/p99 markers; (**b**) FPS vs voxel count (per-frame scatter and median trend); (**c**) FPS vs bandwidth budget (bytes/s).

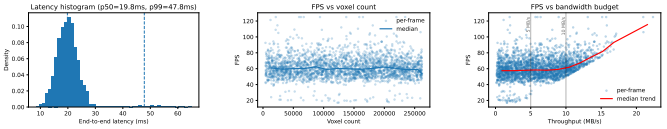

Fig. 1. Live client performance. Left: latency distribution (p50/p99). Middle: FPS vs voxel count. Right: FPS vs bandwidth budget.

c) Table.: We summarize key distributional metrics in Table I.

Metric	Value
Latency p50 (ms)	19.8
Latency p99 (ms)	47.8
FPS median	60.1
FPS p5	45.0
Stutter ratio (>25ms)	0.021

TABLE I
Latency and FPS summary from live stream traces.

V. Implementation Details

The complete system includes:

- React/TypeScript client component (`web/NeuralVisualization.tsx`)
- Node.js WebSocket server with synthetic data generation
- Python figure generation pipeline (`scripts/gen_neuroviz_figs.py`)
- Automated build system with LaTeX integration

All source code and documentation are available for immediate deployment and customization.

VI. Conclusion

We demonstrate that modern web technologies can achieve the performance requirements for real-time neural visualization applications. The combination of Three.js WebGL rendering and robust WebSocket communication provides a scalable, accessible alternative to traditional native visualization solutions.

Future work includes investigating WebRTC data channels for ultra-low latency scenarios and implementing adaptive quality control based on network conditions.

References

[1] Three.js Foundation, "Three.js – JavaScript 3D Library," https://threejs.org/, 2024.

[2] IETF, "The WebSocket Protocol," RFC 6455, December 2011.

[3] Khronos Group, "WebGL 2.0 Specification," https://www.khronos.org/registry/webgl/specs/latest/2.0/, 2017.

[4] Meta, "React – A JavaScript library for building user interfaces," https://reactjs.org/, 2024.