ChatGPT

Spectra Mesh Visualizer Dashboard Implementation

Overview

The **Spectra Mesh Visualizer** is implemented as an interactive web dashboard using HTML, CSS, and JavaScript (D3.js for the visualization). It displays a real-time **mesh network** of system nodes (agents) with unique symbolic representations for each archetype (e.g. **Uniphi**, **Kobalt**, **Phil Mo-Tsu**). The design emphasizes visually rich, layered interactivity and modularity, allowing integration with CLI tools or digital agents via a backend API if needed. Key features include:

- **Real-Time Mesh Canvas:** A dynamic network graph (rendered with D3/SVG) shows nodes and links, updating at ~30 FPS. Each node's appearance (shape/color) denotes its archetype, and continuous entropy changes are reflected as pulsing size/color.
- **Symbolic Node Archetypes:** Different node types use distinctive symbols (circle, square, star) and base colors (e.g. cyan for Uniphi, blue for Kobalt, orange for Phil Mo-Tsu) to provide intuitive visual metaphors.
- **Interactive Nodes:** Users can click on any node to see its **identity and status** in a sidebar. Node details (ID, type, current entropy value, connection count, status level) update in real-time. Nodes can also be dragged to reposition them, with a force-directed layout automatically adjusting connected nodes.
- **Sidebar and HUD:** A sidebar on the right shows the **Node Details** of the selected node and a **Stream Log** of live events. The HUD (heads-up display) overlay in the main canvas shows global status (total nodes, average entropy, overall network state) updated continuously.
- **Live Entropy Streams:** Each node has an entropy value that fluctuates over time (simulated here as a random walk). These entropy "streams" are visualized by node pulsation (size/brightness) and generate event messages (e.g. *entropy spike* alerts) in the log. The stream logic is modular – in a real deployment this could hook into actual data feeds (e.g. via WebSocket or REST API) instead of the built-in simulator.
- **Modular & Extensible:** The code is organized into separate HTML, CSS, and JS files for clarity. It can run as a static web page, or be served from a Node.js/Express server. The logic is data-driven – new node types or data streams can be integrated by adjusting configuration sections. An optional Node.js component (not required for basic usage) could push real data to the front-end (for example, using a WebSocket to broadcast node states/entropy from a CLI agent backend).

Below, we provide the full bundle structure: **HTML**, **CSS**, and **JavaScript** for the dashboard. These files can be saved and zipped as a package (e.g. *SpectraMeshVisualizer.zip*) for easy deployment. Opening the `index.html` file will launch the interactive dashboard.

File Structure

- **index.html** – The main HTML page containing the layout (visualization canvas, sidebar, HUD) and including external scripts/styles.
- **style.css** – Stylesheet for layout and visual theme (dark background, node/link styling, sidebar and HUD formatting).
- **main.js** – JavaScript logic for generating the network, running the D3 force simulation, handling interactions, and simulating real-time entropy updates and events.
- *(Optional)* **server.js** – (Not included here) In a real setup, a Node.js script could feed live data to this front-end (for example, via WebSocket or periodic REST calls). The front-end code is prepared to receive external updates by replacing or augmenting the simulation loop.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Spectra Mesh Visualizer</title>
  <!-- Include D3.js library for the visualization -->
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <div id="container">
    <!-- Visualization Canvas -->
    <div id="viz"></div>
    <!-- Sidebar for node details and stream log -->
    <div id="sidebar">
      <h2>Node Details</h2>
      <div id="nodeDetails">Click on a node to see details</div>
      <h2>Stream Log</h2>
      <div id="streamLog"></div>
    </div>
    <!-- Heads-up Display (HUD) for overall status -->
    <div id="hud"></div>
  </div>
  <script src="main.js"></script>
</body>
</html>
```

style.css

```
/* Base styles for the dashboard */
html, body {
```

```
  margin: 0;
  padding: 0;
  background: #000;
  color: #fff;
  font-family: sans-serif;
  overflow: hidden;
}
#container {
  display: flex;
  flex-direction: row;
  height: 100vh;
  width: 100%;
}
/* Main visualization area */
#viz {
  flex: 1;
  position: relative;
  /* Dark background with a subtle radial glow for visual richness */
  background: radial-gradient(circle at 50% 50%, #111 0%, #000 80%);
}
/* Sidebar styles */
#sidebar {
  width: 250px;
  background: #111;
  border-left: 2px solid #333;
  padding: 10px;
  overflow-y: auto;
}
#sidebar h2 {
  font-size: 1.1em;
  margin: 0.5em 0;
  border-bottom: 1px solid #444;
}
#nodeDetails {
  font-size: 0.9em;
  line-height: 1.4em;
  margin-bottom: 1em;
}
#streamLog {
  font-size: 0.85em;
  line-height: 1.4em;
  max-height: 30vh;
  overflow-y: auto;
  border-top: 1px solid #444;
  padding-top: 0.5em;
}
/* HUD style */
#hud {
```

```
  position: absolute;
  top: 5px;
  left: 5px;
  background: rgba(0,0,0,0.5);
  padding: 3px 8px;
  font-size: 0.8em;
  border-radius: 3px;
}
/* Graph element styles */
.links .link {
  stroke: #888;
  stroke-width: 1px;
  opacity: 0.8;
}
.node path {
  stroke: #fff;
  stroke-width: 1px;
}
```

main.js

```
// Spectra Mesh Visualizer main script

// Create data for nodes and links in the mesh
const nodeTypes = ["Uniphi", "Kobalt", "Phil Mo-Tsu"];
const nodes = [];
const links = [];
let idCounter = 1;

// Create multiple nodes for each archetype (for example, 5 of each type)
nodeTypes.forEach(type => {
  for (let i = 0; i < 5; i++) {
    // Each node gets a unique ID (e.g., "U1", "U2" for Uniphi) and a random
initial entropy
    nodes.push({
      id: type.substring(0, 1) + idCounter,
      type: type,
      entropy: Math.random()
    });
    idCounter++;
  }
});

// Generate random links between nodes to form a mesh network
for (let i = 0; i < nodes.length; i++) {
  for (let j = i + 1; j < nodes.length; j++) {
```

```
    if (Math.random() < 0.15) {  // 15% chance of linking any two nodes
      links.push({ source: nodes[i].id, target: nodes[j].id });
    }
  }
}
// Ensure each node has at least one link (no isolated nodes)
nodes.forEach(node => {
  const connected = links.some(l => l.source === node.id || l.target ===
node.id);
  if (!connected && nodes.length > 1) {
    // Connect this isolated node to a random other node
    let other = nodes[Math.floor(Math.random() * nodes.length)];
    if (other.id === node.id) {
      // if the random pick is itself, choose the next node
      other = nodes[(nodes.indexOf(node) + 1) % nodes.length];
    }
    links.push({ source: node.id, target: other.id });
  }
});

// Set up the SVG canvas dimensions (full window minus sidebar width)
const width = window.innerWidth - 250;
const height = window.innerHeight;
const svg = d3.select("#viz").append("svg")
  .attr("width", width)
  .attr("height", height);

// Define an arrow marker for link arrows (for aesthetics or direction
indication)
svg.append("defs").append("marker")
  .attr("id", "arrow")
  .attr("viewBox", "0 -5 10 10")
  .attr("refX", 15)              // position of the arrow on the line
  .attr("refY", 0)
  .attr("markerWidth", 6)
  .attr("markerHeight", 6)
  .attr("orient", "auto")
  .append("path")
    .attr("d", "M0,-5L10,0L0,5")  // arrowhead shape (triangle)
    .attr("fill", "#888");

// Draw links (lines connecting nodes)
const linkElems = svg.append("g")
    .attr("class", "links")
  .selectAll("line")
  .data(links)
  .enter().append("line")
    .attr("class", "link")
```

```
    .attr("marker-end", "url(#arrow)");  // add arrowhead to end of each line

// Draw node elements (as groups containing a shape)
const nodeElems = svg.append("g")
    .attr("class", "nodes")
  .selectAll("g")
  .data(nodes)
  .enter().append("g")
    .attr("class", "node");

// Use D3 symbol shapes to represent different node archetypes
const symbolScale = d3.scaleOrdinal()
  .domain(nodeTypes)
  .range([d3.symbolCircle, d3.symbolSquare, d3.symbolStar]);  // map each type
to a symbol

nodeElems.append("path")
  .attr("d", d3.symbol().size(200).type(d => symbolScale(d.type)))
  .attr("fill", d => {
    // Base color per type for differentiation
    if (d.type === "Uniphi")       return "#6AFCFC";   // bright cyan
    if (d.type === "Kobalt")       return "#809CFF";   // soft cobalt blue
    if (d.type === "Phil Mo-Tsu")  return "#FFB570";   // orange-peach
    return "#888";
  });

// Add a tooltip (title) on each node for quick identification
nodeElems.append("title")
  .text(d => `${d.id} (${d.type})`);

// Initialize the force simulation for the nodes
const simulation = d3.forceSimulation(nodes)
  .force("link", d3.forceLink(links).id(d => d.id).distance(80))
  .force("charge", d3.forceManyBody().strength(-200))   // repulsive force for
spacing
  .force("center", d3.forceCenter(width / 2, height / 2))
  .force("collision", d3.forceCollide().radius(20));    // avoid nodes getting
too close

// On each simulation tick, update link and node positions
simulation.on("tick", () => {
  linkElems
    .attr("x1", d => d.source.x)
    .attr("y1", d => d.source.y)
    .attr("x2", d => d.target.x)
    .attr("y2", d => d.target.y);
  nodeElems
    .attr("transform", d => `translate(${d.x}, ${d.y})`);
```

```
});

// Drag interaction – enable dragging nodes
const drag = d3.drag()
  .on("start", (event, d) => {
    if (!event.active) simulation.alphaTarget(0.3).restart();  // reheat
simulation
    d.fx = d.x;
    d.fy = d.y;
  })
  .on("drag", (event, d) => {
    d.fx = event.x;
    d.fy = event.y;
  })
  .on("end", (event, d) => {
    if (!event.active) simulation.alphaTarget(0);
    d.fx = null;
    d.fy = null;
  });
nodeElems.call(drag);

// Click interaction – display clicked node's details in the sidebar
let selectedNode = null;
nodeElems.on("click", (event, d) => {
  selectedNode = d;
  updateNodeDetails(d);
});

// Function to update the Node Details panel for a given node
function updateNodeDetails(node) {
  const detailsDiv = document.getElementById("nodeDetails");
  if (!node) {
    detailsDiv.innerHTML = "Click on a node to see details";
  } else {
    // Determine a simple status description based on entropy level
    let statusText = "Nominal";
    if      (node.entropy > 0.9)  statusText = "Critical";
    else if (node.entropy > 0.75) statusText = "High Entropy";
    else if (node.entropy < 0.25) statusText = "Stable";
    // Count this node's connections (degree)
    const connCount = links.filter(l =>
      (l.source.id || l.source) === node.id ||
      (l.target.id || l.target) === node.id
    ).length;
    detailsDiv.innerHTML =
      `<strong>ID:</strong> ${node.id}<br>
       <strong>Type:</strong> ${node.type}<br>
       <strong>Entropy:</strong> ${node.entropy.toFixed(3)}<br>
```

```
        <strong>Status:</strong> ${statusText}<br>
        <strong>Connections:</strong> ${connCount}`;
  }
}

// Function to update the HUD status display
function updateHUD() {
  const hudDiv = document.getElementById("hud");
  const avgEntropy = d3.mean(nodes, d => d.entropy);
  let networkStatus = "Stable";
  if (avgEntropy > 0.75) networkStatus = "Highly Active";
  if (nodes.some(n => n.entropy > 0.9)) networkStatus = "Critical Activity";
  hudDiv.textContent =
    `Nodes: ${nodes.length} | Avg Entropy: ${avgEntropy.toFixed(2)} | Status: $
{networkStatus}`;
}

// Utility to append a message to the stream log
const logDiv = document.getElementById("streamLog");
function addLog(message) {
  const time = new Date().toLocaleTimeString();
  const entry = document.createElement("div");
  entry.textContent = `[${time}] ${message}`;
  logDiv.appendChild(entry);
  // Auto-scroll to latest message
  logDiv.scrollTop = logDiv.scrollHeight;
  // Limit log size to last 100 messages
  if (logDiv.children.length > 100) {
    logDiv.removeChild(logDiv.children[0]);
  }
}

// Real-time entropy update loop (30 FPS approximately)
setInterval(() => {
  // Randomly adjust each node's entropy (simulate entropy stream)
  nodes.forEach(node => {
    const delta = (Math.random() - 0.5) * 0.1;  // small random delta
    node.entropy = Math.max(0, Math.min(1, node.entropy + delta));  // keep in
[0,1]
  });
  // Update node visuals based on entropy
  nodeElems.select("path")
    .attr("transform", d => `scale(${0.8 + d.entropy * 0.5})`)  // pulse size
(scale 0.8–1.3)
    .attr("fill", d => {
      // Brighten the base color toward white as entropy rises (visual "glow")
      if (d.type === "Uniphi")      return d3.interpolateRgb("#6AFCFC",
"#FFFFFF")(d.entropy);
```

```
      if (d.type === "Kobalt")      return d3.interpolateRgb("#809CFF",
"#FFFFFF")(d.entropy);
      if (d.type === "Phil Mo-Tsu") return d3.interpolateRgb("#FFB570",
"#FFFFFF")(d.entropy);
      return "#888";
    })
    .attr("stroke-width", d => 1 + d.entropy * 2);  // thicken outline with
entropy
  // Update the HUD and selected node details every frame
  updateHUD();
  if (selectedNode) {
    updateNodeDetails(selectedNode);
  }
}, 1000 / 30);

// Event stream simulation (log entropy-related events every second)
setInterval(() => {
  // Find the node with highest entropy
  const highNode = nodes.reduce((prev, curr) =>
    curr.entropy > (prev ? prev.entropy : 0) ? curr : prev, null);
  if (highNode && highNode.entropy > 0.8) {
    // Log a spike event for high entropy
    addLog(`Entropy spike on node ${highNode.id} (entropy=$
{highNode.entropy.toFixed(2)})`);
  } else {
    // Occasionally log a nominal status update if no high entropy
    if (Math.random() < 0.3) {
      addLog("Network nominal – no anomalies detected");
    }
  }
}, 1000);
```

Usage: Open `index.html` in a web browser to launch the dashboard. You will see a network graph with nodes labeled by type. Nodes will gently pulse in size/color as their entropy values change. Click on a node to view its details in the sidebar; try dragging nodes to rearrange the graph. The HUD in the top-left displays the total nodes, average entropy, and a simple overall status (e.g. *Stable* vs *Critical Activity*), while the Stream Log on the right shows event messages (e.g. high entropy spikes or periodic status updates).

Integration & Extension: The current implementation simulates entropy streams internally. For a real deployment, you can integrate a Node.js backend or CLI agent output as follows:

- Use a WebSocket or REST API to feed real metrics into the front-end. For example, a Node.js server could broadcast updates for node entropy or new events. The front-end `main.js` can replace the `setInterval` simulation with event handlers that update `nodes` data based on incoming messages, then call `updateHUD()` and `updateNodeDetails()` accordingly.

- The modular structure allows adding new node archetypes by updating the `nodeTypes` array and providing a symbol/color mapping. Likewise, additional data (e.g. CPU load, memory, etc.) could be shown by extending the node objects and tweaking the detail panel and visuals.
- The entire bundle (HTML, CSS, JS) can be zipped for distribution. Ensure that all files are in the same directory when unzipped. If serving via Node.js, place these files in a static public folder or use a simple Express server to serve `index.html` and associated assets.

This implementation provides a rich, interactive foundation for the Spectra Mesh Visualizer. You can further enhance it with custom metaphors (e.g. background starfields, animated link flows, or agent-specific icons) and connect it to live data streams to monitor and control your system's agents in real time. The result is a scalable, visually engaging dashboard that meets the requirements and can evolve with your project. Enjoy exploring the mesh visualizer!
