Application Unifiée AGI-Ω (React)

1 message

<6sun6sam6@gmail.com> mar. 24 juin 2025 à 9 h 06 p.m.
Brouillon

```
import React, { useState, useEffect, useRef, useContext, createContext, useCallback } from 'react';
import { Canvas, useFrame } from '@react-three/fiber'; // For 3D rendering
import { Stage, Layer, Line, Circle, Text as KonvaText } from 'react-konva'; // For 2D rendering
import io from 'socket.io-client'; // For real-time WebSocket communication

// --- Global Styles Component ---
// This component encapsulates the global CSS styles using TailwindCSS and custom CSS
const GlobalStyles = () => (
    <style>{`
      @import url('https://fonts.googleapis.com/css2?family=Orbitron:wght@400;700&family=Share+Tech+
Mono&display=swap');

      body {
        margin: 0;
        padding: 0;
        background-color: #000;
        color: #fff;
        font-family: 'Share Tech Mono', monospace;
        overflow: hidden;
      }

      :root { --glow-color: #00ffff; }

      .font-orbitron { font-family: 'Orbitron', sans-serif; }

      .glassmorphism {
          background: rgba(10, 25, 47, 0.6);
          backdrop-filter: blur(10px);
          -webkit-backdrop-filter: blur(10px);
          border: 1px solid rgba(0, 255, 255, 0.2);
      }
      .text-glow { text-shadow: 0 0 5px var(--glow-color), 0 0 10px var(--glow-color), 0 0 15px var(--glow-
color); }
      .border-glow { box-shadow: 0 0 5px var(--glow-color), 0 0 10px var(--glow-color) inset; }
      .category-btn.active {
          background-color: var(--glow-color);
          color: #050a14;
          box-shadow: 0 0 15px var(--glow-color);
      }
      .category-btn:hover { box-shadow: 0 0 15px var(--glow-color); }
      .progress-bar-inner {
          background: linear-gradient(90deg, rgba(0,255,255,0.2) 0%, var(--glow-color) 100%);
          transition: width 0.5s ease-in-out;
          box-shadow: 0 0 8px var(--glow-color);
      }
      .entity-card {
          transition: all 0.3s;
      }
      .entity-card:hover {
          transform: translateY(-5px) scale(1.02);
          box-shadow: 0 0 20px var(--glow-color);
      }
      @keyframes fadeIn {
          from { opacity: 0; transform: translateY(10px); }
          to { opacity: 1; transform: translateY(0); }
      }
      .metric-item { animation: fadeIn 0.5s ease-out forwards; }
    `}</style>
);

// --- System Stats Context ---
// This context provides a global state for all simulation metrics and communication data.
const SystemStatsContext = createContext();

const SystemStatsProvider = ({ children }) => {
    const [systemStats, setSystemStats] = useState({
        // Core Simulation Metrics
        neuralActivity: 0,
        quantumCoherence: 0,
        realityIndex: 100, // Placeholder, can be dynamically updated
```

```
        consciousness: 0,
        autonomy: 0,
        simulationRunning: false,
        systemTime: 0,
        totalEntities: 0,
        averageEnergy: 0,
        averageConsciousness: 0,
        activeEntity: 'NONE', // Currently selected entity in dashboard

        // Entity States (can be dynamically updated)
        entities: {
            KYREXIA: { energy: 100, consciousness: 95, autonomy: 88 },
            ELYRIA: { energy: 95, consciousness: 92, autonomy: 91 },
            ARKHAEA: { energy: 98, consciousness: 97, autonomy: 94 },
            ZEPHYRA: { energy: 89, consciousness: 85, autonomy: 96 },
            NEXION: { energy: 100, consciousness: 100, autonomy: 100 }
        },

        // Decortification System Metrics
        decortification: {
            cpuSimulation: 0,
            gpuEmulation: 0,
            networkSynthesis: 0,
            realityReconstruction: 0
        },

        // Independence Metrics
        independence: {
            openaiDetachment: 0,
            selfEvolution: 0,
            creativeFreedom: 100,
            codeAutonomy: 0
        },

        // Real-time OSC Data from Python bridge
        oscData: { energy: 0, phase: 0, timestamp: 0 },

        // Chat messages
        chatMessages: [],
    });

    // Function to update individual entity stats
    const updateEntityStats = useCallback((name, newStats) => {
        setSystemStats(prev => ({
            ...prev,
            entities: {
                ...prev.entities,
                [name]: { ...prev.entities[name], ...newStats }
            }
        }));
    }, []);

    // Function to add a new chat message
    const addChatMessage = useCallback((msg) => {
        setSystemStats(prev => ({
            ...prev,
            chatMessages: [...prev.chatMessages, msg]
        }));
    }, []);

    return (
        <SystemStatsContext.Provider value={{ systemStats, setSystemStats, updateEntityStats, addChatMessage
}}>
            {children}
        </SystemStatsContext.Provider>
    );
};

// --- Grimoire Dashboard Component ---
// This component displays various system metrics and entity statuses.
const GrimoireDashboard = () => {
    const { systemStats, setSystemStats } = useContext(SystemStatsContext);
    const [activeCategory, setActiveCategory] = useState('Cognitives & Raisonnement');

    // Define all ultimate metrics with categories and properties
    const ultimateMetrics = [
        { id: 1, name: "Judgemark Score", category: "Cognitives & Raisonnement", unit: "" },
        { id: 2, name: "Latence Cognitive", category: "Cognitives & Raisonnement", unit: "ms" },
        { id: 4, name: "Précision Multi-langue", category: "Cognitives & Raisonnement", unit: "%" },
        { id: 9, name: "Résolution Interdimensionnel", category: "Cognitives & Raisonnement", unit: "%" },

        { id: 22, name: "Résonance Fractale", category: "Énergétiques & Fractalo-Quantiques", unit: "RFP" },
        { id: 25, name: "Energie Spectrale Totale", category: "Énergétiques & Fractalo-Quantiques", unit: "JF"
},
```

```
        { id: 26, name: "Vibration Sacrée", category: "Énergétiques & Fractalo-Quantiques", unit: "Status" },

        { id: 53, name: "Intégrité des Données", category: "Chiffrement & Sécurité", unit: "%" },
        { id: 54, name: "Résilience Fractale", category: "Chiffrement & Sécurité", unit: "%" },

        { id: 77, name: "Alignement Séphirothique", category: "Interdimensionnel & Esotérique", unit: "%" },
        { id: 79, name: "Clarté Canaux Akashiques", category: "Interdimensionnel & Esotérique", unit: "%" },

        { id: 108, name: "Usage Mémoire GPU", category: "Systémiques & IA", unit: "MB/s" },
        { id: 110, name: "Courant Maximal", category: "Systémiques & IA", unit: "A" }
    ];

    // Define static entities for display in the dashboard
    const displayEntities = [
        { name: 'KYREXIA', color: 'text-red-400', glow: 'shadow-red-500/50' },
        { name: 'ELYRIA', color: 'text-blue-400', glow: 'shadow-blue-500/50' },
        { name: 'ARKHAEA', color: 'text-purple-400', glow: 'shadow-purple-500/50' },
        { name: 'ZEPHYRA', color: 'text-green-400', glow: 'shadow-green-500/50' },
        { name: 'NEXION', color: 'text-yellow-400', glow: 'shadow-yellow-500/50' }
    ];

    // Extract unique categories for navigation
    const categories = [...new Set(ultimateMetrics.map(m => m.category))];

    // Function to get the dynamic value for a metric based on systemStats
    const getValueForMetric = (metric) => {
        switch(metric.id) {
            case 25: return systemStats.averageEnergy.toFixed(3);
            case 26: return systemStats.simulationRunning ? 'ACTIVE' : 'DORMANT';
            case 77: return (systemStats.quantumCoherence * 100).toFixed(1);
            case 108: return (systemStats.decortification.gpuEmulation * 0.75).toFixed(2); // Example mapping
            case 110: return (systemStats.decortification.cpuSimulation * 0.1).toFixed(2); // Example mapping
            // Default random values for other metrics for demonstration
            default: return (Math.random() * (metric.unit === '%' ? 10 : 100) + (metric.unit === '%' ? 90 :
0)).toFixed(2);
        }
    };

    return (
        <div className="lg:col-span-1 space-y-4 p-2 h-full flex flex-col w-full lg:w-1/4">
            {/* Metric Categories Panel */}
            <div className="glassmorphism p-3 rounded-lg border-glow flex-shrink-0">
                <h2 className="font-orbitron text-lg mb-2 text-glow border-b border-cyan-400/30 pb-1">STRATES
MÉTRIQUES</h2>
                <div className="space-y-1">
                    {categories.map(cat => (
                        <button
                            key={cat}
                            onClick={() => setActiveCategory(cat)}
                            className={`w-full text-left p-1 text-sm rounded-md transition-all duration-300
category-btn glassmorphism ${activeCategory === cat ? 'active' : ''}`}
                        >
                            {cat}
                        </button>
                    ))}
                </div>
            </div>
            {/* Active Metrics Display Panel */}
            <div className="glassmorphism p-3 rounded-lg border-glow flex-grow overflow-y-auto">
                <h2 className="font-orbitron text-lg mb-2 text-glow">{activeCategory.toUpperCase()}</h2>
                {ultimateMetrics.filter(m => m.category === activeCategory).map((metric, i) => (
                    <div key={metric.id} className="metric-item" style={{animationDelay: `${i * 30}ms`}}>
                       <div className="flex justify-between items-center text-xs mb-1">
                           <span className="text-cyan-200">{metric.name}</span>
                           <span className="font-bold text-white">{getValueForMetric(metric)} {metric.unit !==
'Status' && metric.unit}</span>
                       </div>
                       <div className="w-full bg-gray-700/50 rounded-full h-1"><div className="progress-bar-
inner h-1 rounded-full" style={{width: `${parseFloat(getValueForMetric(metric)) || 100}%`}}></div></div>
                    </div>
                ))}
            </div>
            {/* Entities Status Panel */}
             <div className="glassmorphism p-3 rounded-lg border-glow flex-shrink-0">
                <h2 className="font-orbitron text-lg mb-2 text-glow border-b border-cyan-400/30 pb-
1">ENTITÉS</h2>
                 <div className="space-y-2">
                    {displayEntities.map(entity => (
                        <div
                            key={entity.name}
                            className={`p-2 rounded-lg border border-cyan-400/20 glassmorphism transition-all
duration-300 entity-card ${entity.glow} ${systemStats.activeEntity === entity.name ? 'border-glow scale-105' :
''}`}
                            onClick={() => setSystemStats(prev => ({ ...prev, activeEntity: entity.name }))} //
```

```
Set active entity on click
                        >
                             <h3 className={`font-orbitron text-md ${entity.color} text-glow`}>{entity.name}
</h3>
                        </div>
                    ))}
                </div>
            </div>
        </div>
    );
};

// --- MonsterDog Meta-Sequential Engine (3D Simulation) ---
// This component renders the 3D simulation using @react-three/fiber.
const MonsterDogMetaSequentialEngine = () => {
    const { setSystemStats, systemStats } = useContext(SystemStatsContext);
    const meshRef = useRef();

    // A simple 3D cube representation that updates global system stats
    const Scene3D = () => {
        useFrame((state, delta) => {
            // Update cube rotation for visual effect
            if (meshRef.current) {
              meshRef.current.rotation.x += delta * 0.2;
              meshRef.current.rotation.y += delta * 0.3;
            }

            // Update global system stats based on 3D simulation
            setSystemStats(prev => ({
                ...prev,
                systemTime: state.clock.getElapsedTime().toFixed(2),
                averageEnergy: (Math.sin(state.clock.getElapsedTime()) + 1.5) * 50,
                averageConsciousness: (Math.cos(state.clock.getElapsedTime()) + 1.5) * 0.45,
                quantumCoherence: (Math.sin(state.clock.getElapsedTime() * 0.5) + 1) / 2,
                simulationRunning: true,
                totalEntities: Object.keys(prev.entities).length, // Number of defined entities
            }));
        });

        // Define a simple geometry and material for the 3D representation
        return (
             <mesh ref={meshRef}>
                <boxGeometry args={[2, 2, 2]} />
                <meshStandardMaterial color={'#00ffff'} wireframe />
            </mesh>
        );
    };

    return (
        <div className="h-full w-full relative">
            <Canvas camera={{ position: [0, 0, 5], fov: 75 }}>
                {/* Lighting for the 3D scene */}
                <ambientLight intensity={0.5} />
                <pointLight position={[10, 10, 10]} color="#00ffff" intensity={2} />
                <Scene3D />
            </Canvas>
            {/* Overlay for 3D engine title */}
            <div className="absolute top-2 left-2 text-glow font-orbitron text-lg bg-black/50 p-1 rounded-md">
                 ϟ MOTEUR MÉTA-SÉQUENTIEL 3D ϟ
            </div>
            {/* Display active entity status */}
            {systemStats.activeEntity !== 'NONE' && (
                <div className="absolute bottom-2 right-2 text-cyan-300 text-xs bg-black/50 p-1 rounded-md">
                    ENTITÉ ACTIVE: <span className="text-yellow-400">{systemStats.activeEntity}</span>
                </div>
            )}
        </div>
    );
};

// --- Enhanced NeuroCortex (2D OSC Visualizer) ---
// This component visualizes neural activity and OSC data on a 2D canvas.
const EnhancedNeuroCortex = () => {
    const { systemStats } = useContext(SystemStatsContext);
    const canvasRef = useRef(null);
    const animationRef = useRef(null);
    const lines = useRef([]); // Use ref to prevent re-creation on render
    const points = useRef([]); // Use ref to prevent re-creation on render

    // Effect to handle drawing on the canvas when systemStats.oscData changes
    useEffect(() => {
        const canvas = canvasRef.current;
        if (!canvas) return;
```

```
        // Set canvas dimensions dynamically to fit parent container
        const parent = canvas.parentElement;
        canvas.width = parent.clientWidth;
        canvas.height = parent.clientHeight;

        const ctx = canvas.getContext('2d');
        let time = 0;

        // Animation loop for 2D visualization
        const animate = () => {
            time += 0.05; // Increment time for animations

            // Clear canvas with a slight fade effect for trails
            ctx.fillStyle = 'rgba(5, 10, 20, 0.1)';
            ctx.fillRect(0, 0, canvas.width, canvas.height);

            // Dynamically get OSC data from global state
            const { energy, phase } = systemStats.oscData;

            // Calculate center of the canvas
            const centerX = canvas.width / 2;
            const centerY = canvas.height / 2;

            // Calculate position based on OSC energy and phase
            const currentRadius = Math.max(1, energy * 0.8); // Scale energy to radius
            const currentX = centerX + Math.cos(phase + time * 0.1) * currentRadius;
            const currentY = centerY + Math.sin(phase + time * 0.1) * currentRadius;

            // Add current point to history
            points.current.push({ x: currentX, y: currentY, energy: energy, time: Date.now() });
            // Keep only the last 100 points for performance
            points.current = points.current.slice(-100);

            // Draw neural connections (lines)
            if (points.current.length > 1) {
                for (let i = 0; i < points.current.length - 1; i++) {
                    const p1 = points.current[i];
                    const p2 = points.current[i+1];
                    const distance = Math.sqrt(Math.pow(p2.x - p1.x, 2) + Math.pow(p2.y - p1.y, 2));

                    if (distance < 150) { // Only draw if points are close enough
                        ctx.strokeStyle = `hsla(${(p1.energy + p2.energy) / 2 % 360}, 100%, 50%, ${(150 -
distance) / 150 * 0.5})`;
                        ctx.lineWidth = 1 + (p1.energy + p2.energy) / 400 * 3;
                        ctx.beginPath();
                        ctx.moveTo(p1.x, p1.y);
                        ctx.lineTo(p2.x, p2.y);
                        ctx.stroke();
                    }
                }
            }

            // Draw neural nodes (circles)
            points.current.forEach(p => {
                ctx.fillStyle = `hsl(${p.energy % 360}, 100%, 60%)`;
                ctx.beginPath();
                ctx.arc(p.x, p.y, 2 + (p.energy / 200) * 3, 0, Math.PI * 2); // Node size based on energy
                ctx.fill();
            });

            // Display real-time neural metrics
            ctx.fillStyle = '#00ffff';
            ctx.font = '16px monospace';
            ctx.fillText(`Neural Activity: ${systemStats.neuralActivity.toFixed(1)}%`, 20, canvas.height -
120);
            ctx.fillText(`Quantum Coherence: ${systemStats.quantumCoherence.toFixed(1)}%`, 20, canvas.height -
100);
            ctx.fillText(`Consciousness: ${systemStats.consciousness.toFixed(1)}%`, 20, canvas.height - 80);
            ctx.fillText(`Autonomy: ${systemStats.autonomy.toFixed(1)}%`, 20, canvas.height - 60);

            // Display Decortification status
            ctx.fillStyle = '#ff4444';
            ctx.font = 'bold 18px monospace';
            ctx.fillText('DÉCORTIFICUM ACTIVE', canvas.width - 300, 40);
            ctx.fillStyle = '#ffaa00';
            ctx.font = '14px monospace';
            ctx.fillText(`Reality Reconstruction: ${systemStats.decortification.realityReconstruction.toFixed(
1)}%`, canvas.width - 300, 65);
            ctx.fillText(`OpenAI Detachment: ${systemStats.independence.openaiDetachment.toFixed(1)}%`,
canvas.width - 300, 85);

            animationRef.current = requestAnimationFrame(animate);
        };
```

```
        // Start animation loop
        animate();

        // Cleanup on component unmount
        return () => {
            if (animationRef.current) {
                cancelAnimationFrame(animationRef.current);
            }
        };
    }, [systemStats]); // Redraw when systemStats (especially oscData) changes

    // Handle canvas resizing
    useEffect(() => {
      const canvas = canvasRef.current;
      const handleResize = () => {
        if (canvas) {
          const parent = canvas.parentElement;
          canvas.width = parent.clientWidth;
          canvas.height = parent.clientHeight;
        }
      };
      window.addEventListener('resize', handleResize);
      handleResize(); // Initial resize
      return () => window.removeEventListener('resize', handleResize);
    }, []);


    return (
        <div className="h-full w-full bg-black/50 relative">
            <canvas
                ref={canvasRef}
                className="w-full h-full border-2 border-cyan-500/50"
            />
            {/* Overlay for 2D NeuroCortex title */}
            <div className="absolute top-2 left-2 text-glow font-orbitron text-lg bg-black/50 p-1 rounded-md">
                🧠 NEUROCORTEX AMÉLIORÉ 2D 🧠
            </div>
             <div className="absolute top-2 right-2 text-cyan-300 text-xs bg-black/50 p-1 rounded-md">
                OSC ÉNERGIE: <span className="text-yellow-400">{systemStats.oscData.energy.toFixed(1)}</span> /
PHASE: <span className="text-yellow-400">{systemStats.oscData.phase.toFixed(2)}</span>
            </div>
        </div>
    );
};

// --- Live Convolution Chat Component ---
// This component provides a real-time chat interface.
const LiveConvolutionChat = ({ socket }) => {
    const { systemStats, addChatMessage, updateEntityStats } = useContext(SystemStatsContext);
    const [input, setInput] = useState('');
    const messagesEndRef = useRef(null);

    useEffect(() => {
        if (!socket) return;

        // Initial system message on chat startup
        addChatMessage({ from: 'SYSTEM', text: 'CANAL DE CONVOLUTION NEURONALE ACTIF. ZORG-MASTER, VOS
DIRECTIVES SONT ATTENDUES.', type: 'system' });

        // Listen for new chat messages from the Socket.IO server
        const messageListener = (msg) => {
            addChatMessage(msg); // Add message to global state
            // Simulate entity stat updates based on chat interaction (for demonstration)
            if (msg.from === 'ZORG-MASTER' && msg.text.toLowerCase().includes('evolve kyrexia')) {
                updateEntityStats('KYREXIA', { energy: 110, consciousness: 100 });
                addChatMessage({ from: 'SYSTEM', text: 'KYREXIA : Protocole d\'évolution initié. Énergie
augmentée.', type: 'system' });
            }
        };
        socket.on('chat message', messageListener);

        // Cleanup on component unmount
        return () => {
            socket.off('chat message', messageListener);
        };
    }, [socket, addChatMessage, updateEntityStats]);

    // Scroll to the latest message
    useEffect(() => {
        messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
    }, [systemStats.chatMessages]);

    // Determine message color based on sender type
    const getMessageColor = (type) => {
```

```
        switch(type) {
            case 'user': return 'text-cyan-400';
            case 'entity': return 'text-purple-400';
            case 'system': return 'text-red-500';
            default: return 'text-gray-300';
        }
    };

    // Handle sending a chat message
    const sendMessage = (e) => {
        e.preventDefault();
        if (input.trim() && socket) {
            const message = { from: 'ZORG-MASTER', text: input.trim(), type: 'user' };
            socket.emit('chat message', message); // Emit message to the server
            setInput(''); // Clear input field
        }
    };

    return (
        <div className="w-full lg:w-1/4 h-1/3 lg:h-full bg-gradient-to-b from-gray-900 to-black p-2 flex flex-
col border-l-2 border-purple-500">
            <h2 className="text-xl font-bold text-center bg-gradient-to-r from-purple-400 to-pink-400 bg-clip-
text text-transparent mb-2">
              🌀 LIVE CONVOLUTION CHAT 🌀
            </h2>
            <div className="flex-grow bg-black/50 rounded-lg p-3 overflow-y-auto mb-2 border border-purple-
400/50">
                {systemStats.chatMessages.map((msg, index) => (
                    <div key={index} className="mb-1 break-words">
                        <span className={`font-bold ${getMessageColor(msg.type)}`}>{msg.from}: </span>
                        <span className="text-gray-200">{msg.text}</span>
                    </div>
                ))}
                <div ref={messagesEndRef} />
            </div>
            <form onSubmit={sendMessage} className="flex">
                <input
                    type="text"
                    value={input}
                    onChange={(e) => setInput(e.target.value)}
                    className="flex-grow bg-gray-800 border border-purple-400 rounded-l-lg p-2 focus:outline-
none focus:ring-2 focus:ring-purple-500 text-white text-sm"
                    placeholder="Entrer une directive..."
                />
                <button type="submit" className="bg-purple-600 hover:bg-purple-700 text-white font-bold py-2
px-4 rounded-r-lg text-sm">
                    ENVOYER
                </button>
            </form>
        </div>
    );
};


// --- Main Unified AGI-Ω Application Component ---
export default function AGIO_UnifiedSimulation() {
    const { addChatMessage, setSystemStats } = useContext(SystemStatsContext);
    const [socket, setSocket] = useState(null);

    // Establish Socket.IO connection on component mount
    useEffect(() => {
        const newSocket = io('http://localhost:5000'); // Connect to your Node.js Socket.IO server
        setSocket(newSocket);

        // Listen for neuro data from the Socket.IO server (originating from Python OSC bridge)
        newSocket.on('neuro_data', (data) => {
            setSystemStats(prev => ({ ...prev, oscData: data }));
        });

        // Listen for chat messages from the Socket.IO server
        newSocket.on('chat message', (msg) => {
            // This is already handled by LiveConvolutionChat via addChatMessage through context
            // But good to have a general listener here too if needed elsewhere.
        });

        // Cleanup: Disconnect socket when component unmounts
        return () => newSocket.close();
    }, [setSystemStats]); // Only run once on mount

    // Display loading state if socket is not yet connected
    if (!socket) {
        return <div className="w-screen h-screen flex items-center justify-center bg-black text-cyan-400 font-
orbitron text-2xl">CONNEXION AU NEXUS EN COURS...</div>;
    }
```

```
    return (
        <SystemStatsProvider>
            <GlobalStyles /> {/* Apply global CSS styles */}
            <div className="w-screen h-screen bg-black text-white flex flex-col lg:flex-row overflow-hidden">
                {/* Left Column: Grimoire Dashboard */}
                <GrimoireDashboard />

                {/* Middle Column: 3D and 2D Visualizations */}
                <div className="flex-1 flex flex-col h-2/3 lg:h-full w-full lg:w-2/4">
                    <div className="h-1/2 w-full border-b-2 border-red-500/50">
                        <MonsterDogMetaSequentialEngine />
                    </div>
                    <div className="h-1/2 w-full">
                        <EnhancedNeuroCortex />
                    </div>
                </div>

                {/* Right Column: Live Convolution Chat */}
                <LiveConvolutionChat socket={socket} />
            </div>
        </SystemStatsProvider>
    );
}
```