ChatGPT

Extended Cosmic Simulation Code

Below is a complete HTML file with the extended cosmic simulation, incorporating **Interference**, **Ratio**, **Chaos**, **Entropy**, and an **Energy/Information/Wave/Particle** conversion model. New UI controls and parameters have been added and integrated into the WebGL shaders and simulation logic as described. Comments in the code highlight the implementation of each new feature.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Cosmic Simulation – Extended Uniphi Model</title>
  <!-- Style for simulation canvas and UI, preserving existing aesthetic -->
  <style>
    *, *::before, *::after { box-sizing: border-box; }
    html, body { margin: 0; padding: 0; width: 100%; height: 100%; overflow:
hidden;
                 background: #000; font: 14px/1.4 system-ui, Helvetica, Arial,
sans-serif; color: #fff; }
    canvas { display: block; position: fixed; inset: 0; width: 100%; height:
100%; }
    #ui { position: fixed; top: 0; left: 0; max-height: 100vh; overflow-y: auto;
          width: 380px; background: rgba(0,0,0,0.55); backdrop-filter:
blur(8px);
          padding: 16px 20px; border-right: 1px solid hsla(0,0%,100%,0.12); z-
index: 10; }
    h2 { margin: 0 0 10px; font-size: 16px; font-weight: 600; text-transform:
uppercase; letter-spacing: 0.04em; }
    details { margin-bottom: 14px; }
    summary { cursor: pointer; font-weight: 600; margin: 6px 0; }
    .ctrl { display: flex; flex-direction: column; margin-bottom: 6px; }
    .ctrl label { font-size: 12px; opacity: 0.85; display: flex; justify-
content: space-between; }
    .ctrl input[type="range"], .ctrl input[type="number"] { width: 100%; }
    .ctrl input[type="checkbox"] { margin-right: 4px; }
    button { background: #1e90ff; border: none; color: #fff; padding: 6px 12px;
             border-radius: 4px; font-size: 13px; cursor: pointer; margin-top:
6px; width: 100%; }
  </style>
</head>
<body>
  <!-- Canvas layers for various simulation modes -->
  <canvas id="gl"></canvas>         <!-- Orbital particles layer -->
```

```
  <canvas id="deepField" style="display:none"></canvas>
<!-- Deep-field waves layer -->
  <canvas id="gravityWave" style="display:none"></canvas>  <!-- Gravitational
wave overlay -->
  <canvas id="lensQuad" style="display:none"></canvas>     <!-- Gravitational
lens layer (unused in this demo) -->
  <canvas id="blackHoleLayer" style="display:none"></canvas>   <!-- Black hole
layer (for compatibility) -->
  <canvas id="multiverseLayer" style="display:none"></canvas>  <!-- Multiverse
layer (for compatibility) -->
  <canvas id="bioLayer" style="display:none"></canvas>     <!-- Bio layer (for
compatibility) -->
  <canvas id="emergentLayer" style="display:none"></canvas><!-- Emergent layer
(for compatibility) -->
  <canvas id="overlay" style="display:none"></canvas>      <!-- Overlay (not
used here) -->

  <!-- UI Control Panel -->
  <div id="ui">
    <h2>Simulation Controls</h2>
    <!-- Master render mode selection -->
    <details open>
      <summary>Render Mode</summary>
      <div class="ctrl"><label><input name="mode" type="radio" value="particles"
checked> Particles (Orbital)</label></div>
      <div class="ctrl"><label><input name="mode" type="radio" value="waves">
Deep-Field Waves</label></div>
      <div class="ctrl"><label><input name="mode" type="radio" value="gwave">
Gravity Waves</label></div>
    </details>

    <!-- Deep-Field forces toggles (for wave interference context) -->
    <details>
      <summary>Deep-Field Forces</summary>
      <div class="ctrl"><label><input id="df_grav" type="checkbox" checked>
Gravity</label></div>
      <div class="ctrl"><label><input id="df_em" type="checkbox" checked>
Electromagnetism</label></div>
      <div class="ctrl"><label><input id="df_strong" type="checkbox"> Strong</
label></div>
      <div class="ctrl"><label><input id="df_weak" type="checkbox"> Weak</
label></div>
    </details>

    <!-- Interference control -->
    <details>
      <summary>Interference</summary>
      <div class="ctrl">
```

```
        <label>Interference Intensity <span id="interferenceVal">0.00</span></
label>
        <input id="interferenceSlider" type="range" min="0" max="1" step="0.01"
value="0">
      </div>
    </details>

    <!-- Physical ratios controls -->
    <details>
      <summary>Physical Ratios</summary>
      <div class="ctrl">
        <label>Mass/Velocity Ratio <span id="ratioVal">1.0</span></label>
        <input id="ratio" type="range" min="0" max="2" step="0.1" value="1">
      </div>
      <div class="ctrl">
        <label>Density <span id="densityVal">200</span></label>
        <input id="density" type="range" min="50" max="300" step="10"
value="200">
      </div>
    </details>

    <!-- Chaos control -->
    <details>
      <summary>Chaos</summary>
      <div class="ctrl">
        <label>Chaos Level <span id="chaosVal">0.00</span></label>
        <input id="chaos" type="range" min="0" max="1" step="0.01" value="0">
      </div>
    </details>

    <!-- Entropy control -->
    <details>
      <summary>Entropy</summary>
      <div class="ctrl">
        <label><input id="entropyToggle" type="checkbox"> Enable Entropy
Increase</label>
      </div>
    </details>

    <!-- Energy/Information/Wave/Particle conversion model controls -->
    <details>
      <summary>Energy/Information/Wave/Particle</summary>
      <div class="ctrl">Energy: <span id="energyVal">100.0</span></div>
      <div class="ctrl">Information: <span id="infoVal">50.0</span></div>
      <div class="ctrl">Waves: <span id="waveVal">25.0</span></div>
      <div class="ctrl">Particles: <span id="particleVal">75.0</span></div>
      <div class="ctrl">
        <label>Conversion Rate <span id="convRateVal">0.10</span></label>
```

```
        <input id="convRate" type="range" min="0" max="1" step="0.01"
value="0.1">
      </div>
    </details>

    <!-- Gravitational waves parameters (amplitude & frequency) -->
    <details>
      <summary>Gravity Waves</summary>
      <div class="ctrl">
        <label>Amplitude <span id="gwAmpVal">1.0</span></label>
        <input id="gw_amp" type="range" min="0" max="5" step="0.1" value="1">
      </div>
      <div class="ctrl">
        <label>Frequency <span id="gwFreqVal">1.0</span></label>
        <input id="gw_freq" type="range" min="0.1" max="10" step="0.1"
value="1">
      </div>
    </details>
  </div>

  <!-- Main simulation script (with embedded shaders and logic) -->
  <script type="module">
  'use strict';
  // Simulation parameters (global state)
  const Params = {
    mode: 'particles',
    deep:  { grav: true, em: true, strong: false, weak: false, interference:
0.0 },  // deep-field forces and interference factor
    lens:  { mass: 1.1, k:
1.0 },       // lens distortion parameters (unused in this extension)
    gwave: { amplitude: 1.0, frequency: 1.0 },  // gravitational wave parameters
 1
    ratio: { massVel: 1.0, density: 200 },      // physical ratios:
gravitational (mass-velocity) and particle count
    chaos: 0.0,                     // chaos intensity level
    entropyEnabled: false           // entropy mode toggle
  };

  // Conversion model state
  const convState = {
    energy: 100.0,
    info: 50.0,
    wave: 25.0,
    particle: 75.0
  };
  let convRate = 0.1;  // conversion rate (fraction per second)

  // Canvas and WebGL context handles
```

```
  const cvs = {};
  let oGL, dGL, gCtx;
  // WebGL programs and buffers
  let oProg, oPosBuf, oColBuf;
  let dProg, dBuf, dLoc;
  // Orbital bodies array
  let oBodies = [], oLastTime = 0;
  // Last timestamp for conversion updates
  let convLastTime = 0;

  // Utility: get canvas by id
  const $C = id => document.getElementById(id);

  window.addEventListener('DOMContentLoaded', () => {
    // Setup canvas references
    cvs.gl   = $C('gl');
    cvs.deep = $C('deepField');
    cvs.gwave= $C('gravityWave');
    cvs.lens = $C('lensQuad');
    cvs.overlay = $C('overlay');
    cvs.bio  = $C('bioLayer');
    cvs.emerg= $C('emergentLayer');
    // Initialize canvas contexts and WebGL programs
    initOrbital();
    initDeepField();
    initGravityWaves();
    // (Lens, bio, emerg layers remain uninitialized in this extension)

    // Mode switch (radio buttons)
    document.querySelectorAll('[name="mode"]').forEach(radio => {
      radio.addEventListener('input', () => {
        Params.mode = radio.value;
        updateLayers();
      });
    });

    // Deep-field force toggles
    [['df_grav','grav'], ['df_em','em'], ['df_strong','strong'],
['df_weak','weak']].forEach(([id, key]) => {
      const el = $C(id);
      if (el) el.addEventListener('input', () => {
        Params.deep[key] = el.checked;
      });
    });

    // Generic binder for sliders (updates param and text span)
    const bind = (id, branch, key, spanID) => {
      const el = $C(id), span = $C(spanID);
```

```
    el.addEventListener('input', () => {
      Params[branch][key] = parseFloat(el.value);
      if (span) span.textContent = el.value;
    });
  };
  // Bind new feature sliders
  bind('interferenceSlider', 'deep', 'interference', 'interferenceVal');
  bind('ratio', 'ratio', 'massVel', 'ratioVal');
  bind('density', 'ratio', 'density', 'densityVal');
  bind('gw_amp', 'gwave', 'amplitude', 'gwAmpVal');
  bind('gw_freq', 'gwave', 'frequency', 'gwFreqVal');

  // Chaos slider
  const chaosSlider = $C('chaos');
  const chaosValSpan = $C('chaosVal');
  chaosSlider.addEventListener('input', () => {
    Params.chaos = parseFloat(chaosSlider.value);
    chaosValSpan.textContent = chaosSlider.value;
  });

  // Entropy toggle
  const entropyToggle = $C('entropyToggle');
  entropyToggle.addEventListener('change', () => {
    Params.entropyEnabled = entropyToggle.checked;
  });

  // Conversion rate slider
  const convSlider = $C('convRate');
  const convValSpan = $C('convRateVal');
  convSlider.addEventListener('input', () => {
    convRate = parseFloat(convSlider.value);
    convValSpan.textContent = convSlider.value;
  });

  // Begin the animation loop
  requestAnimationFrame(function loop(time) {
    // Update conversion model each frame (regardless of mode)
    updateConversion(time);
    // Render active mode
    if (Params.mode === 'particles') drawOrbital(time);
    if (Params.mode === 'waves')    drawDeepField(time);
    if (Params.mode === 'gwave')    drawGravityWaves(time);
    // (Other modes like lens, etc., are not utilized in this extension)
    requestAnimationFrame(loop);
  });
});

// Show/hide canvases based on current mode 2
```

```
  function updateLayers() {
    cvs.gl.style.display    = (Params.mode === 'particles') ? 'block' : 'none';
    cvs.deep.style.display  = (Params.mode === 'waves') ? 'block' : 'none';
    cvs.gwave.style.display = (Params.mode === 'gwave') ? 'block' : 'none';
    if (cvs.lens)   cvs.lens.style.display   = 'none';
    if (cvs.bio)    cvs.bio.style.display    = 'none';
    if (cvs.emerg)  cvs.emerg.style.display  = 'none';
    if (cvs.overlay) cvs.overlay.style.display = 'none';  // overlay not used
  }

  // Initialize orbital layer (particle cluster with gravitational interaction)
  function initOrbital() {
    oGL = cvs.gl.getContext('webgl2', {antialias: true}) ||
cvs.gl.getContext('webgl');
    if (!oGL) {
      console.error('WebGL not supported for orbital canvas');
      return;
    }
    // Adjust canvas for device pixel ratio
    const resize = () => {
      const dpr = window.devicePixelRatio || 1;
      cvs.gl.width = innerWidth * dpr;
      cvs.gl.height = innerHeight * dpr;
      oGL.viewport(0, 0, cvs.gl.width, cvs.gl.height);
    };
    window.addEventListener('resize', resize);
    resize();
    // Create shader program for point rendering (simple pass-through with
color)
    const vsSource = `#version 300 es
      layout(location=0) in vec2 a_pos;
      layout(location=1) in vec3 a_col;
      out vec3 vCol;
      void main() {
        vCol = a_col;
        gl_Position = vec4(a_pos, 0.0, 1.0);
        gl_PointSize = 3.5;
      }`;
    const fsSource = `#version 300 es
      precision mediump float;
      in vec3 vCol;
      out vec4 fragColor;
      void main() {
        float d = length(gl_PointCoord - 0.5);
        if (d > 0.5) discard;
        fragColor = vec4(vCol, 1.0);
      }`;
```

```
  const compileShader = (src, type) => {
    const sh = oGL.createShader(type);
    oGL.shaderSource(sh, src);
    oGL.compileShader(sh);
    return sh;
  };
  oProg = oGL.createProgram();
  oGL.attachShader(oProg, compileShader(vsSource, oGL.VERTEX_SHADER));
  oGL.attachShader(oProg, compileShader(fsSource, oGL.FRAGMENT_SHADER));
  oGL.linkProgram(oProg);
  if (!oGL.getProgramParameter(oProg, oGL.LINK_STATUS)) {
    console.error(oGL.getProgramInfoLog(oProg));
  }
  oGL.useProgram(oProg);
  // Create buffers for positions and colors
  oPosBuf = oGL.createBuffer();
  oColBuf = oGL.createBuffer();
  oGL.enableVertexAttribArray(0);
  oGL.enableVertexAttribArray(1);
  oGL.bindBuffer(oGL.ARRAY_BUFFER, oPosBuf);
  oGL.vertexAttribPointer(0, 2, oGL.FLOAT, false, 0, 0);
  oGL.bindBuffer(oGL.ARRAY_BUFFER, oColBuf);
  oGL.vertexAttribPointer(1, 3, oGL.FLOAT, false, 0, 0);
  // Define body class for particles
  const rand = (a=0, b=1) => Math.random() * (b - a) + a;
  class Body {
    constructor() {
      this.pos = [ rand(-1, 1), rand(-1, 1) ];
      const angle = Math.random() * 2.0 * Math.PI;
      const speed = rand(0.001, 0.003);
      this.vel = [ Math.cos(angle) * speed, Math.sin(angle) * speed ];
      this.col = [ Math.random(), Math.random(), Math.random() ];
    }
  }
  // Initialize bodies array with default count
  createBodies(Params.ratio.density);
  oLastTime = performance.now();
}

// Helper to create a new bodies array of given size
function createBodies(count) {
  oBodies = Array.from({ length: count }, () => new Body());
}

// Orbital layer draw (update physics and render particles) 5 6
function drawOrbital(time) {
  if (!oGL) return;
  // Calculate time step
```

```
    if (!oLastTime) oLastTime = time;
    const dt = Math.min(0.033, (time - oLastTime) / 1000);
    oLastTime = time;
    const Gbase = 0.2;
    const G = Gbase *
Params.ratio.massVel;  // adjust gravitational constant by mass/velocity ratio
slider
    const N = oBodies.length;
    // N-body gravity interactions (O(N^2))
    for (let i = 0; i < N; i++) {
      const A = oBodies[i];
      for (let j = i + 1; j < N; j++) {
        const B = oBodies[j];
        const dx = B.pos[0] - A.pos[0];
        const dy = B.pos[1] - A.pos[1];
        const dist2 = dx*dx + dy*dy + 0.001;
        const invDist = 1.0 / Math.sqrt(dist2);
        const force = G / dist2;
        // Accelerate A and B toward each other
        const fx = force * dx * invDist;
        const fy = force * dy * invDist;
        A.vel[0] += fx * dt;
        A.vel[1] += fy * dt;
        B.vel[0] -= fx * dt;
        B.vel[1] -= fy * dt;
      }
    }
    // Chaos: inject random velocity perturbations based on chaos slider
    if (Params.chaos > 0) {
      const noiseStrength = 0.05;
      for (const b of oBodies) {
        b.vel[0] += (Math.random() - 0.5) * 2 * Params.chaos * noiseStrength *
dt;
        b.vel[1] += (Math.random() - 0.5) * 2 * Params.chaos * noiseStrength *
dt;
      }
    }
    // Entropy: if enabled, gradually increase disorder (equalize velocities and
diffuse colors)
    if (Params.entropyEnabled) {
      // Compute average speed (for energy dispersion) and average color (for
color diffusion)
      let avgSpeed = 0;
      let avgR = 0, avgG = 0, avgB = 0;
      for (const b of oBodies) {
        const speed = Math.hypot(b.vel[0], b.vel[1]);
        avgSpeed += speed;
        avgR += b.col[0];
```

```
      avgG += b.col[1];
      avgB += b.col[2];
    }
    avgSpeed /= oBodies.length;
    avgR /= oBodies.length;
    avgG /= oBodies.length;
    avgB /= oBodies.length;
    const equalizeRate = 0.5;   // rate per second to equalize speeds
    const diffuseRate  = 0.5;   // rate per second to diffuse colors
    for (const b of oBodies) {
      // Adjust velocity magnitude toward average speed (energy distribution)
      const v = b.vel;
      const speed = Math.hypot(v[0], v[1]);
      if (speed > 0) {
        const newMag = speed + (avgSpeed - speed) * (equalizeRate * dt);
        const scale = newMag / speed;
        b.vel[0] *= scale;
        b.vel[1] *= scale;
      }
      // Diffuse color toward average color (color diffusion)
      b.col[0] += (avgR - b.col[0]) * (diffuseRate * dt);
      b.col[1] += (avgG - b.col[1]) * (diffuseRate * dt);
      b.col[2] += (avgB - b.col[2]) * (diffuseRate * dt);
    }
  }
  // Update positions based on velocities
  for (const b of oBodies) {
    b.pos[0] += b.vel[0] * dt;
    b.pos[1] += b.vel[1] * dt;
  }
  // Entropy: add slight random position drift (spatial disorder) if enabled
  if (Params.entropyEnabled) {
    const posNoise = 0.02;
    for (const b of oBodies) {
      b.pos[0] += (Math.random() - 0.5) * 2 * posNoise * dt;
      b.pos[1] += (Math.random() - 0.5) * 2 * posNoise * dt;
    }
  }
  // If density (particle count) slider changed, adjust particle count
  const desiredCount = Math.floor(Params.ratio.density);
  if (desiredCount !== oBodies.length) {
    createBodies(desiredCount);
  }
  // Prepare position and color buffers
  const positions = new Float32Array(oBodies.length * 2);
  const colors    = new Float32Array(oBodies.length * 3);
  oBodies.forEach((b, i) => {
    positions[2*i]   = b.pos[0];
```

```
      positions[2*i+1] = b.pos[1];
      colors[3*i]      = b.col[0];
      colors[3*i+1]    = b.col[1];
      colors[3*i+2]    = b.col[2];
    });
    oGL.useProgram(oProg);
    oGL.bindBuffer(oGL.ARRAY_BUFFER, oPosBuf);
    oGL.bufferData(oGL.ARRAY_BUFFER, positions, oGL.DYNAMIC_DRAW);
    oGL.bindBuffer(oGL.ARRAY_BUFFER, oColBuf);
    oGL.bufferData(oGL.ARRAY_BUFFER, colors, oGL.DYNAMIC_DRAW);
    // Clear and draw particles
    oGL.clearColor(0, 0, 0, 1);
    oGL.clear(oGL.COLOR_BUFFER_BIT);
    oGL.drawArrays(oGL.POINTS, 0, oBodies.length);
  }

  // Initialize deep-field layer (WebGL shader for wave field) 7
  function initDeepField() {
    dGL = cvs.deep.getContext('webgl2') || cvs.deep.getContext('webgl');
    if (!dGL) {
      console.error('WebGL not supported for deep field');
      return;
    }
    // Resize canvas to viewport
    const resize = () => {
      cvs.deep.width = innerWidth;
      cvs.deep.height = innerHeight;
      dGL.viewport(0, 0, cvs.deep.width, cvs.deep.height);
    };
    window.addEventListener('resize', resize);
    resize();
    // Vertex shader (full-screen quad)
    const vsSource = `attribute vec4 a_position; void main() { gl_Position =
a_position; }`;
    // Fragment shader for deep-field forces and interference patterns
    const fsSource = `precision mediump float;
      uniform float u_time;
      uniform vec2 u_resolution;
      uniform int u_gravity;
      uniform int u_em;
      uniform int u_strong;
      uniform int u_weak;
      uniform float u_interference;
      // Compute field strength from fundamental forces toggles 8
      float fieldStrength(vec2 uv) {
        float s = 0.0;
        if (u_gravity == 1)  s += 0.3 / length(uv);
        if (u_em == 1)       s += 0.3 * sin(10.0 * uv.x) * cos(10.0 * uv.y);
```

```
        if (u_strong == 1)   s += 0.5 * exp(-10.0 * length(uv));
        if (u_weak == 1)     s += 0.2 * sin(30.0 * length(uv));
        return s;
      }
      void main() {
        // Normalized coordinates
        vec2 uv = (gl_FragCoord.xy - 0.5 * u_resolution.xy) / u_resolution.y;
        // Base field strength from toggled forces
        float base = fieldStrength(uv);
        // Wave interference: combine two wave patterns if interference > 0
        float wave1 = sin(10.0 * uv.x) * cos(10.0 * uv.y);
        float wave2 = sin((10.0 + 5.0 * u_interference) * uv.x) * cos((10.0 +
5.0 * u_interference) * uv.y);
        // Mix original wave pattern with a second wave based on interference
intensity
        float interferencePattern = mix(wave1, 0.5 * (wave1 + wave2),
u_interference);
        // If EM waves are on, replace their effect with interference pattern
        float s = base;
        if (u_em == 1) {
          // Remove original EM contribution and use interference pattern scaled
similarly
          s += 0.3 * (interferencePattern - sin(10.0 * uv.x) * cos(10.0 *
uv.y));
        }
        // Compute dynamic color from field strength and time
        vec3 color = vec3(
          sin(s * 2.0 + u_time),
          cos(s + u_time),
          sin(s * 0.5)
        );
        gl_FragColor = vec4(color * 0.5 + 0.5, 1.0);
      }`;
    // Compile shaders and link program
    const compileShader = (src, type) => {
      const shader = dGL.createShader(type);
      dGL.shaderSource(shader, src);
      dGL.compileShader(shader);
      if (!dGL.getShaderParameter(shader, dGL.COMPILE_STATUS)) {
        console.error(dGL.getShaderInfoLog(shader));
      }
      return shader;
    };
    dProg = dGL.createProgram();
    const vsh = compileShader(vsSource, dGL.VERTEX_SHADER);
    const fsh = compileShader(fsSource, dGL.FRAGMENT_SHADER);
    dGL.attachShader(dProg, vsh);
    dGL.attachShader(dProg, fsh);
```

```
  dGL.linkProgram(dProg);
  if (!dGL.getProgramParameter(dProg, dGL.LINK_STATUS)) {
    console.error(dGL.getProgramInfoLog(dProg));
  }
  // Get uniform locations
  dLoc = {
    time: dGL.getUniformLocation(dProg, 'u_time'),
    res:  dGL.getUniformLocation(dProg, 'u_resolution'),
    g:    dGL.getUniformLocation(dProg, 'u_gravity'),
    em:   dGL.getUniformLocation(dProg, 'u_em'),
    st:   dGL.getUniformLocation(dProg, 'u_strong'),
    wk:   dGL.getUniformLocation(dProg, 'u_weak'),
    intf: dGL.getUniformLocation(dProg, 'u_interference')
  };
  // Create fullscreen triangle buffer
  dBuf = dGL.createBuffer();
  dGL.bindBuffer(dGL.ARRAY_BUFFER, dBuf);
  // Two triangles covering the viewport
  const verts = new Float32Array([
    -1, -1,  1, -1,  -1, 1,
    -1,  1,  1, -1,   1, 1
  ]);
  dGL.bufferData(dGL.ARRAY_BUFFER, verts, dGL.STATIC_DRAW);
}

// Draw deep-field waves (with interference) each frame
function drawDeepField(time) {
  if (!dGL) return;
  dGL.useProgram(dProg);
  // Set up vertex attribute
  dGL.bindBuffer(dGL.ARRAY_BUFFER, dBuf);
  const posLoc = dGL.getAttribLocation(dProg, 'a_position');
  dGL.enableVertexAttribArray(posLoc);
  dGL.vertexAttribPointer(posLoc, 2, dGL.FLOAT, false, 0, 0);
  // Pass uniforms
  dGL.uniform1f(dLoc.time, time * 0.001);
  dGL.uniform2f(dLoc.res, cvs.deep.width, cvs.deep.height);
  dGL.uniform1i(dLoc.g,  Params.deep.grav ? 1 : 0);
  dGL.uniform1i(dLoc.em, Params.deep.em ? 1 : 0);
  dGL.uniform1i(dLoc.st, Params.deep.strong ? 1 : 0);
  dGL.uniform1i(dLoc.wk, Params.deep.weak ? 1 : 0);
  dGL.uniform1f(dLoc.intf, Params.deep.interference);
  // Render full-screen triangles
  dGL.drawArrays(dGL.TRIANGLES, 0, 6);
}

// Initialize gravitational wave overlay (2D canvas for wave ripples)
function initGravityWaves() {
```

```
  gCtx = cvs.gwave.getContext('2d');
  if (!gCtx) return;
  // Size canvas to viewport
  const resize = () => {
    cvs.gwave.width = innerWidth;
    cvs.gwave.height = innerHeight;
  };
  window.addEventListener('resize', resize);
  resize();
  // gCtx will be used in drawGravityWaves()
}

// Draw gravitational wave pattern on 2D overlay 9
function drawGravityWaves(time) {
  if (!gCtx) return;
  const W = cvs.gwave.width, H = cvs.gwave.height;
  gCtx.clearRect(0, 0, W, H);
  const a = Params.gwave.amplitude;
  const f = Params.gwave.frequency;
  // Draw vertical strip pattern simulating wave interference ripples
  for (let x = 0; x < W; x += 2) {
    const y = Math.sin(x * f + time * 0.001) * a * (H * 0.5) + H * 0.5;
    gCtx.fillStyle = '#fff';
    gCtx.fillRect(x, y, 1, 1);
  }
}

// Update energy-information-wave-particle conversion model
function updateConversion(time) {
  if (!convLastTime) convLastTime = time;
  const dt = (time - convLastTime) / 1000;
  convLastTime = time;
  if (dt <= 0 || convRate <= 0) {
    // Just update display without changes
    $C('energyVal').textContent    = convState.energy.toFixed(1);
    $C('infoVal').textContent      = convState.info.toFixed(1);
    $C('waveVal').textContent      = convState.wave.toFixed(1);
    $C('particleVal').textContent  = convState.particle.toFixed(1);
    return;
  }
  // Simulate conversions: energy -> wave -> particle -> info -> energy loop
  const k = convRate * dt;
  // Compute transfers for this step
  const dE = -k * convState.energy + k * convState.info;
  const dI = -k * convState.info + k * convState.particle;
  const dW = -k * convState.wave + k * convState.energy;
  const dP = -k * convState.particle + k * convState.wave;
  convState.energy    += dE;
```

```
    convState.info      += dI;
    convState.wave      += dW;
    convState.particle  += dP;
    // Update displayed values (rounded to one decimal)
    $C('energyVal').textContent    = convState.energy.toFixed(1);
    $C('infoVal').textContent      = convState.info.toFixed(1);
    $C('waveVal').textContent      = convState.wave.toFixed(1);
    $C('particleVal').textContent  = convState.particle.toFixed(1);
  }
  </script>
</body>
</html>
```

Sources:

- Adapted from the Spectra Gallery **superuniverse** WebGL simulation (including orbital N-body integration and deep-field shader) [3] [4] [10].
- Gravitational wave overlay and parameters based on the unified simulation guide [11] [9].

---

[1] [9] [11] unified-simulation.md
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/b8817fbd3cd1139a19034fd16475897c9511d1a2/docs/unified-simulation.md

[2] [3] [4] [5] [6] [7] [8] [10] superuniverse.js
https://github.com/spectra-gallery/arquolab-sandbox-models/blob/b8817fbd3cd1139a19034fd16475897c9511d1a2/static/nods/superuniverse.js