ChatGPT

Dynamic Quantum Structural Color Visualizer

This solution creates a single-page WebGL visualization that blends abstract physics with color theory. In **concept**, it treats light waves and particles together: colors are mapped from photon wavelengths (380–750 nm for visible light [1]) and use their wave interactions as a generative driver. Photon energy is inversely related to wavelength (shorter wavelength=higher energy) [2], so the code animates multiple sinusoidal waves to simulate interference patterns akin to structural color in nature [3]. These wave patterns are rendered on an HTML5 canvas using a custom vertex shader and fragment shader pair [4], producing dynamic harmonics of color.

A **WebGL implementation** is used with vanilla JavaScript: we set up a full-screen canvas and compile GLSL shaders. The vertex shader simply passes geometry through, while the fragment shader computes color per pixel based on time and position, combining several sinusoidal "particle wave" components to simulate interference. For example, two orthogonal wave components (sine functions of *u* and *v* coordinates plus time) are added, and their sum is mapped to color. The mapping uses both RGB and HSL domains: users can toggle color modes or pick base hues. HSL is included because it's a common cylindrical color model (hue, saturation, lightness) used in graphics and color pickers [5]. UI elements (HTML `<input>` and `<select>`) control parameters like wave frequencies, speeds, and color mode, allowing a **playful interplay** of settings. The code uses `requestAnimationFrame` for smooth animation, avoiding stale content by continuously updating the canvas each frame.

Key implementation details: - We create an HTML `<canvas id="glcanvas">` and obtain a WebGL context in JS. - A quad covering the viewport is drawn. The **vertex shader** sets `gl_Position` to cover the full screen. The **fragment shader** (GLSL) takes uniforms like time (`u_time`) and user-chosen color parameters, and computes an RGB color.
- To simulate structural coloration, the fragment shader uses wave interference (e.g. `float wave = sin(freq1*(x+y)+time) + sin(freq2*(x-y)-time);`) and maps that to colors.
- The color mapping uses a hue rotation and mixing. We convert between color spaces: in RGB mode we directly set `vec3 color = vec3(r,g,b)`, while in HSL mode we convert from an HSL value to RGB (to give the user intuitive hue/saturation control [5]). - The code includes sliders and color pickers. For example, a `<select>` lets you choose "RGB" or "HSL" mode, and a `<input type="range">` adjusts the wave frequency. Changing these inputs updates shader uniforms in real-time.
- This follows standard WebGL practice: every time we render, we bind the shader program and update uniform values (as per [52]), then draw the geometry.

The complete HTML/JS code is given below. It is a standalone page (no external libraries) and combines WebGL setup, shader source, and UI. The result is a dynamic, colorful visualization that evokes quantum wave–particle duality (Feynman-like paths) with structural color effects and spectrum overlays, all under user control:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>Quantum Structural Color Visualizer</title>
  <style>
    body { margin:0; overflow:hidden; background:#000; color:#eee; font-
family:sans-serif; }
    #ui { position: absolute; top: 10px; left: 10px; z-index: 10; background:
rgba(0,0,0,0.5); padding: 10px; border-radius:8px; }
    label { display:block; margin:5px 0; }
    canvas { display:block; width:100vw; height:100vh; }
  </style>
</head>
<body>
  <div id="ui">
    <label>
      Color Mode:
      <select id="colorMode">
        <option value="rgb">RGB</option>
        <option value="hsl">HSL</option>
      </select>
    </label>
    <label>
      Base Hue (degrees):
      <input type="range" id="hue" min="0" max="360" value="200">
    </label>
    <label>
      Wave Frequency 1:
      <input type="range" id="freq1" min="1" max="10" step="0.5" value="3">
    </label>
    <label>
      Wave Frequency 2:
      <input type="range" id="freq2" min="1" max="10" step="0.5" value="4">
    </label>
    <label>
      Time Speed:
      <input type="range" id="speed" min="0" max="5" step="0.1" value="1.0">
    </label>
  </div>
  <canvas id="glcanvas"></canvas>
  <script>
    // Setup WebGL context
    const canvas = document.getElementById('glcanvas');
    const gl = canvas.getContext('webgl');
    if (!gl) { alert('WebGL not supported'); throw 'WebGL not supported'; }
    // Adjust size
    function resize() {
      canvas.width = window.innerWidth;
      canvas.height = window.innerHeight;
```

```
  gl.viewport(0, 0, gl.drawingBufferWidth, gl.drawingBufferHeight);
}
window.addEventListener('resize', resize);
resize();

// Vertex shader: full-screen quad
const vsSource = `
  attribute vec2 a_position;
  void main() {
    gl_Position = vec4(a_position, 0.0, 1.0);
  }
`;
// Fragment shader: wave interference color
const fsSource = `
  precision mediump float;
  uniform float u_time;
  uniform float u_freq1, u_freq2, u_speed;
  uniform vec3 u_baseColor; // base hue color in RGB
  uniform int u_colorMode;  // 0 = RGB, 1 = HSL
  uniform vec2 u_resolution;
  // Convert HSL to RGB
  vec3 hsl2rgb(float h, float s, float l) {
    float c = (1.0 - abs(2.0*l - 1.0)) * s;
    float x = c * (1.0 - abs(mod(h/60.0, 2.0) - 1.0));
    float m = l - c/2.0;
    vec3 rgb;
    if (h < 60.0) rgb = vec3(c,x,0);
    else if (h < 120.0) rgb = vec3(x,c,0);
    else if (h < 180.0) rgb = vec3(0,c,x);
    else if (h < 240.0) rgb = vec3(0,x,c);
    else if (h < 300.0) rgb = vec3(x,0,c);
    else rgb = vec3(c,0,x);
    return rgb + vec3(m);
  }
  void main() {
    // Normalize coordinates to [-1,1]
    vec2 uv = (gl_FragCoord.xy / u_resolution)*2.0 - 1.0;
    // Aspect ratio correction
    uv.x *= u_resolution.x / u_resolution.y;
    // Compute two wave patterns
    float t = u_time * u_speed;
    float wave1 = sin((uv.x + uv.y) * u_freq1 + t);
    float wave2 = sin((uv.x - uv.y) * u_freq2 - t);
    float waveMix = wave1 + wave2;
    // Normalize to 0..1
    float intensity = (waveMix * 0.5) + 0.5;
    // Base color from user selection (already an RGB color or HSL hue)
    vec3 color;
```

```
        if (u_colorMode == 0) {
          // RGB mode: use base color and modulate brightness
          color = u_baseColor * (0.5 + 0.5 * intensity);
        } else {
          // HSL mode: interpret baseColor.r as hue in [0,360]
          float hue = u_baseColor.r;
          color = hsl2rgb(hue, 0.6, 0.5 + 0.5*intensity);
        }
        // Apply a spectral overlay: mix with a rainbow gradient based on screen
position
        float spectrum = uv.y * 0.5 + 0.5;  // normalized vertical
        vec3 rainbow = hsl2rgb(360.0*spectrum, 0.5, 0.5);
        // Blend spectrum as an overlay
        color = mix(color, rainbow, 0.3 * intensity);
        gl_FragColor = vec4(color, 1.0);
      }
    `;

    // Compile shader function
    function compileShader(src, type) {
      const shader = gl.createShader(type);
      gl.shaderSource(shader, src);
      gl.compileShader(shader);
      if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        console.error(gl.getShaderInfoLog(shader));
        throw 'Shader compile failed';
      }
      return shader;
    }
    // Link program
    const vs = compileShader(vsSource, gl.VERTEX_SHADER);
    const fs = compileShader(fsSource, gl.FRAGMENT_SHADER);
    const program = gl.createProgram();
    gl.attachShader(program, vs);
    gl.attachShader(program, fs);
    gl.bindAttribLocation(program, 0, 'a_position');
    gl.linkProgram(program);
    if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
      console.error(gl.getProgramInfoLog(program));
      throw 'Program link failed';
    }
    gl.useProgram(program);

    // Fullscreen quad coordinates
    const quadVerts = new Float32Array([
      -1, -1,   1, -1,  -1, 1,
       1, -1,  -1,  1,   1, 1
    ]);
```

```
const quadBuf = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, quadBuf);
gl.bufferData(gl.ARRAY_BUFFER, quadVerts, gl.STATIC_DRAW);
gl.enableVertexAttribArray(0);
gl.vertexAttribPointer(0, 2, gl.FLOAT, false, 0, 0);

// Get uniform locations
const u_timeLoc = gl.getUniformLocation(program, 'u_time');
const u_freq1Loc = gl.getUniformLocation(program, 'u_freq1');
const u_freq2Loc = gl.getUniformLocation(program, 'u_freq2');
const u_speedLoc = gl.getUniformLocation(program, 'u_speed');
const u_baseColorLoc = gl.getUniformLocation(program, 'u_baseColor');
const u_colorModeLoc = gl.getUniformLocation(program, 'u_colorMode');
const u_resolutionLoc = gl.getUniformLocation(program, 'u_resolution');

// UI elements
const colorModeSel = document.getElementById('colorMode');
const hueInput = document.getElementById('hue');
const freq1Input = document.getElementById('freq1');
const freq2Input = document.getElementById('freq2');
const speedInput = document.getElementById('speed');

function render(time) {
  // Update uniforms from UI
  const freq1 = parseFloat(freq1Input.value);
  const freq2 = parseFloat(freq2Input.value);
  const speed = parseFloat(speedInput.value);
  let colorMode = colorModeSel.value === 'hsl' ? 1 : 0;
  let hue = parseFloat(hueInput.value);
  // Compute base color uniform (as RGB or HSL data)
  let baseColor;
  if (colorMode === 0) {
    // RGB mode: convert hue to RGB vector (full saturation, lightness=0.5)
    baseColor = [0.0, 0.0, 0.0];
    // Simple mapping: treat hue slider as hue and convert
    const h = hue;
    const s = 1.0, l = 0.5;
    // Convert HSL to RGB (JS)
    const c = (1 - Math.abs(2*l-1)) * s;
    const x = c * (1 - Math.abs((h/60) % 2 - 1));
    let [r1,g1,b1] = [0,0,0];
    if (h < 60) [r1,g1,b1] = [c,x,0];
    else if (h < 120) [r1,g1,b1] = [x,c,0];
    else if (h < 180) [r1,g1,b1] = [0,c,x];
    else if (h < 240) [r1,g1,b1] = [0,x,c];
    else if (h < 300) [r1,g1,b1] = [x,0,c];
    else [r1,g1,b1] = [c,0,x];
    const m = l - c/2;
```

```
        baseColor = [r1+m, g1+m, b1+m];
      } else {
        // HSL mode: pass hue in baseColor.r, others unused
        baseColor = [hue, 0.0, 0.0];
      }
      // Set uniforms
      gl.uniform1f(u_timeLoc, time * 0.001);
      gl.uniform1f(u_freq1Loc, freq1);
      gl.uniform1f(u_freq2Loc, freq2);
      gl.uniform1f(u_speedLoc, speed);
      gl.uniform3fv(u_baseColorLoc, baseColor);
      gl.uniform1i(u_colorModeLoc, colorMode);
      gl.uniform2f(u_resolutionLoc, canvas.width, canvas.height);
      // Draw
      gl.drawArrays(gl.TRIANGLES, 0, 6);
      requestAnimationFrame(render);
    }
    requestAnimationFrame(render);
  </script>
</body>
</html>
```

This code creates an interactive visualization: moving wave patterns (inspired by quantum particle waves) whose **colors** are derived from spectrum and structural effects. Users can adjust frequencies and color mode in real-time via the UI, giving a creative and dynamic effect. The shaders and scripting follow standard WebGL practices [4], and the color selection (RGB vs. HSL) reflects common digital color models [5]. The resulting render taps into physics-inspired themes (photon waves, interference, structural color) to produce a playful, evolving display.

Sources: Concepts are grounded in optics and graphics: visible light ranges [1], photon energy relations [2], and structural coloration via interference [3]. WebGL implementation follows known patterns [4], and HSL color usage is supported by graphics literature [5].

---

[1] Visible spectrum - Wikipedia
https://en.wikipedia.org/wiki/Visible_spectrum

[2] Photon energy - Wikipedia
https://en.wikipedia.org/wiki/Photon_energy

[3] Structural coloration - Wikipedia
https://en.wikipedia.org/wiki/Structural_coloration

[4] WebGL Shaders and GLSL
https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html

[5] HSL and HSV - Wikipedia
https://en.wikipedia.org/wiki/HSL_and_HSV