```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>The Lab - Enhanced Reflections & Lighting</title>
    <script src="https://cdn.tailwindcss.com"></script>
    <link
href="https://fonts.googleapis.com/css2?family=Inter:wght@400;500;600;
700&display=swap" rel="stylesheet">
    <style>
        html, body {
            margin: 0; padding: 0; overflow: hidden;
            background: #000; color: #fff;
            font-family: 'Inter', Segoe UI, sans-serif;
        }
        canvas { display: block; width: 100%; height: 100%; }
        .control-panel::-webkit-scrollbar { width: 6px; }
        .control-panel::-webkit-scrollbar-thumb { background-color:
rgba(255,255,255,0.3); border-radius: 3px; }
        .control-panel::-webkit-scrollbar-track { background:
rgba(0,0,0,0.2); }
        .glassmorphic-panel {
            background: rgba(30, 30, 45, 0.65);
            backdrop-filter: blur(14px) saturate(180%);
            -webkit-backdrop-filter: blur(14px) saturate(180%);
            border: 1px solid rgba(255, 255, 255, 0.12);
        }
        input[type="range"] {
            -webkit-appearance: none; appearance: none; width: 100%;
height: 6px;
            background: rgba(255, 255, 255, 0.1); border-radius: 3px;
outline: none; transition: background 0.3s;
        }
        input[type="range"]::-webkit-slider-thumb {
            -webkit-appearance: none; appearance: none; width: 16px;
height: 16px;
            background: #cbd5e1; border-radius: 50%; cursor: pointer;
border: 2px solid rgba(30, 30, 45, 0.8);
        }
        input[type="range"]:hover { background: rgba(255, 255, 255,
0.2); }
        .btn-icon {
            background: rgba(255,255,255,0.1);
            border-radius: 50%;
            transition: all 0.2s;
        }
```

```
        .btn-icon:hover {
            background: rgba(255,255,255,0.2);
            transform: scale(1.1);
        }
    </style>
</head>
<body>

    <!-- UI Panels -->
    <div class="absolute top-4 left-4 z-10 flex flex-col gap-4
max-h-[calc(100vh-2rem)]">
        <div id="controlsContainer" class="glassmorphic-panel p-5
rounded-xl shadow-2xl w-full max-w-xs sm:max-w-sm md:max-w-[300px]
control-panel overflow-y-auto">
            <div class="flex justify-between items-center mb-4">
                <h2 class="text-xl font-bold text-white/90">Simulation
Controls</h2>
                <div class="flex items-center gap-2">
                    <button id="playPauseBtn" class="p-2 btn-icon"
title="Pause Physics">
                        <svg id="pauseIcon"
xmlns="http://www.w3.org/2000/svg" class="h-5 w-5" viewBox="0 0 20 20"
fill="currentColor"><path fill-rule="evenodd" d="M18 10a8 8 0 11-16 0
8 8 0 0116 0zM7 8a1 1 0 012 0v4a1 1 0 11-2 0V8zm5-1a1 1 0 00-1 1v4a1 1
0 102 0V8a1 1 0 00-1-1z" clip-rule="evenodd" /></svg>
                        <svg id="playIcon"
xmlns="http://www.w3.org/2000/svg" class="h-5 w-5 hidden" viewBox="0 0
20 20" fill="currentColor"><path fill-rule="evenodd" d="M10 18a8 8 0
100-16 8 8 0 000 16zM9.555 7.168A1 1 0 008 8v4a1 1 0 001.555.832l3-2a1
1 0 000-1.664l-3-2z" clip-rule="evenodd" /></svg>
                    </button>
                    <button id="resetBtn" class="p-2 btn-icon"
title="Reset Simulation">
                        <svg xmlns="http://www.w3.org/2000/svg"
class="h-5 w-5" viewBox="0 0 20 20" fill="currentColor"><path
fill-rule="evenodd" d="M4 2a1 1 0 011 1v2.101a7.002 7.002 0 0111.601
2.566 1 1 0 11-1.885.666A5.002 5.002 0 005.999 7H9a1 1 0 110 2H4a1 1 0
01-1-1V3a1 1 0 011-1zm.008 9.057a1 1 0 011.276.61A5.002 5.002 0
0014.001 13H11a1 1 0 110-2h5a1 1 0 011 1v5a1 1 0 11-2 0v-2.101a7.002
7.002 0 01-11.601-2.566 1 1 0 01.61-1.276z" clip-rule="evenodd"
/></svg>
                    </button>
                </div>
            </div>

            <div id="collapsible-content">
                <h3 class="text-lg font-semibold text-white/90
mb-3">Merger</h3>
```

```
                <div class="mb-5 mt-1">
                    <h4 class="text-sm font-medium text-white/70
mb-2">Simulation Phase</h4>
                    <div class="w-full bg-black/20 rounded-full
h-2.5">
                        <div id="progressBar" class="bg-gradient-to-r
from-sky-400 to-indigo-500 h-2.5 rounded-full transition-all
duration-500" style="width: 0%"></div>
                    </div>
                    <div class="text-center mt-2">
                        <span id="statusText" class="text-lg
font-semibold text-sky-300">Initializing...</span>
                    </div>
                </div>
                <div class="space-y-5">
                    <div>
                        <div class="flex justify-between items-center
text-sm text-white/80 mb-1">
                            <label for="massRatioSlider">Mass Ratio
(M₂/M₁)</label>
                            <span id="massRatioValue">0.80</span>
                        </div>
                        <input type="range" id="massRatioSlider"
min="0.1" max="1.0" step="0.01" value="0.8">
                    </div>
                    <div>
                        <div class="flex justify-between items-center
text-sm text-white/80 mb-1">
                            <label for="separationSlider">Initial
Separation</label>
                            <span id="separationValue">20</span>
                        </div>
                        <input type="range" id="separationSlider"
min="10" max="30" step="1" value="20">
                    </div>
                     <div>
                        <div class="flex justify-between items-center
text-sm text-white/80 mb-1">
                            <label for="timeScaleSlider">Simulation
Speed</label>
                            <span id="timeScaleValue">1.0x</span>
                        </div>
                        <input type="range" id="timeScaleSlider"
min="0.0" max="3.0" step="0.1" value="1.0">
                    </div>
                </div>
            </div>
        </div>
```

```
    </div>

    <script type="importmap">
    {
      "imports": {
        "three":
"https://cdn.jsdelivr.net/npm/three@0.163.0/build/three.module.js",
        "three/addons/":
"https://cdn.jsdelivr.net/npm/three@0.163.0/examples/jsm/"
      }
    }
    </script>
    <script type="module">
        import * as THREE from 'three';
        import { OrbitControls } from
'three/addons/controls/OrbitControls.js';
        import { EffectComposer } from
'three/addons/postprocessing/EffectComposer.js';
        import { RenderPass } from
'three/addons/postprocessing/RenderPass.js';
        import { UnrealBloomPass } from
'three/addons/postprocessing/UnrealBloomPass.js';
        import { ShaderPass } from
'three/addons/postprocessing/ShaderPass.js';

        let scene, camera, renderer, composer, controls, clock;
        let lensingPass, wavePass, bloomPass;
        let mergerFlare;
        let room, water, chromeBowl;
        let cubeCamera;
        let coloredLights = [];

        const sim = {
            physicsPaused: false,
            scenePaused: false,
            timeScale: 1.0,
            phase: 'INSPIRAL',
            phaseTime: 0,
            progress: 0,
            wallWaveAmplitude: 0.0,
            config: { massRatio: 0.8, initialSeparation: 20 },
            bh1: null, bh2: null, mergedBh: null,
        };

        const G = 150;
        const DISK_PARTICLE_COUNT = 15000;
        const MERGER_FLARE_PARTICLES = 5000;
        const C = 30;
```

```
        const ROOM_SIZE = 250;
        const ROOM_HEIGHT = 125;

        class BlackHole {
            constructor(mass, position, velocity, color1, color2) {
                this.mass = mass;
                this.pos = position;
                this.vel = velocity;
                this.radius = Math.pow(this.mass, 0.7) * 0.2;

                this.mesh = this.createMesh(color1);
                this.disk = this.createDisk(color2);
                this.mesh.scale.setScalar(this.radius);
                scene.add(this.mesh, this.disk);
            }

            createMesh(color) {
                const group = new THREE.Group();
                const bhMat = new THREE.MeshBasicMaterial({ color:
0x000000 });
                const bhMesh = new THREE.Mesh(new
THREE.SphereGeometry(1, 64, 64), bhMat);

                const photonMat = new THREE.ShaderMaterial({
                    uniforms: {
                        glowColor: { value: new THREE.Color(color) },
                        viewVector: { value: new THREE.Vector3() }
                    },
                    vertexShader: `varying float intensity; uniform
vec3 viewVector; void main() { vec3 actual_normal =
normalize(normalMatrix * normal); intensity = pow(0.7 -
dot(viewVector, actual_normal), 2.0); gl_Position = projectionMatrix *
modelViewMatrix * vec4(position, 1.0); }`,
                    fragmentShader: `varying float intensity; uniform
vec3 glowColor; void main() { gl_FragColor = vec4(glowColor, 1.0) *
intensity; }`,
                    transparent: true, blending:
THREE.AdditiveBlending, side: THREE.BackSide,
                });
                const photonMesh = new THREE.Mesh(new
THREE.SphereGeometry(1.5, 64, 64), photonMat);
                bhMesh.add(photonMesh);
                group.add(bhMesh);
                return group;
            }

            createDisk() {
                const positions = new Float32Array(DISK_PARTICLE_COUNT
```

```
* 3);
                const colors = new Float32Array(DISK_PARTICLE_COUNT *
3);
                const geometry = new THREE.BufferGeometry();
                geometry.setAttribute('position', new
THREE.BufferAttribute(positions, 3));
                geometry.setAttribute('color', new
THREE.BufferAttribute(colors, 3));

                const material = new THREE.PointsMaterial({
                    size: 0.05, vertexColors: true, transparent: true,
opacity: 0.8,
                    blending: THREE.AdditiveBlending, depthWrite:
false
                });

                const points = new THREE.Points(geometry, material);
                points.userData.particles = [];
                for(let i = 0; i < DISK_PARTICLE_COUNT; i++) {
                    points.userData.particles.push({ pos: new
THREE.Vector3(), vel: new THREE.Vector3(), life: 0, baseRadius: 0,
angle: 0 });
                }
                return points;
            }

            update(dt) {
                this.pos.add(this.vel.clone().multiplyScalar(dt));
                this.mesh.position.copy(this.pos);
                this.disk.position.copy(this.pos);
            }

            setVisible(visible) {
                this.mesh.visible = visible;
                this.disk.visible = visible;
            }
        }

        init();

        function init() {
            clock = new THREE.Clock();
            scene = new THREE.Scene();
            scene.background = new THREE.Color(0x000000);

            camera = new THREE.PerspectiveCamera(60, window.innerWidth
/ window.innerHeight, 0.1, 5000);
            camera.position.set(0, ROOM_HEIGHT / 2, 60);
```

```
            renderer = new THREE.WebGLRenderer({ antialias: true,
powerPreference: 'high-performance' });
            renderer.setPixelRatio(window.devicePixelRatio);
            renderer.setSize(window.innerWidth, window.innerHeight);
            renderer.toneMapping = THREE.ACESFilmicToneMapping;
            renderer.toneMappingExposure = 1.8;
            document.body.appendChild(renderer.domElement);

            const cubeRenderTarget = new
THREE.WebGLCubeRenderTarget(512);
            cubeCamera = new THREE.CubeCamera(1, 1000,
cubeRenderTarget);
            cubeCamera.position.y = ROOM_HEIGHT * 0.4;

            controls = new OrbitControls(camera, renderer.domElement);
            controls.target.set(0, ROOM_HEIGHT / 2, 0);
            controls.enableDamping = true; controls.dampingFactor =
0.04;
            controls.maxDistance = 150;
            controls.minDistance = 10;

            createSceneLights();
            createBackgroundStructure();
            createMergerFlare();
            setupPostProcessing();
            setupUI();

            resetSimulation();
            animate();
        }

        function createSceneLights() {
            const colors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00,
0x4b0082, 0xee82ee];
            const lightIntensity = 150;
            const lightDistance = 200;

            for (let i = 0; i < colors.length; i++) {
                const light = new THREE.PointLight(colors[i],
lightIntensity, lightDistance);
                const angle = (i / colors.length) * Math.PI * 2;
                light.position.set(
                    Math.cos(angle) * ROOM_SIZE * 0.4,
                    ROOM_HEIGHT / 2,
                    Math.sin(angle) * ROOM_SIZE * 0.4
                );
                scene.add(light);
```

```
            coloredLights.push(light);
        }

        // Add two bright white lights inside the bowl to brighten
reflections
        const innerLight1 = new THREE.PointLight(0xffffff, 200,
100);
        innerLight1.position.set(20, ROOM_HEIGHT * 0.4, 20);
        scene.add(innerLight1);
        coloredLights.push(innerLight1);

        const innerLight2 = new THREE.PointLight(0xffffff, 200,
100);
        innerLight2.position.set(-20, ROOM_HEIGHT * 0.4, -20);
        scene.add(innerLight2);
        coloredLights.push(innerLight2);
    }

    function createBackgroundStructure() {
        // Room Box
        room = new THREE.Mesh(
            new THREE.BoxGeometry(ROOM_SIZE * 2, ROOM_HEIGHT * 2,
ROOM_SIZE * 2),
            new THREE.MeshBasicMaterial({ color: 0x000000, side:
THREE.BackSide })
        );
        scene.add(room);

        // Reflective Chrome Bowl
        const bowlGeo = new THREE.SphereGeometry(ROOM_SIZE / 2,
64, 32);
        const chromeMat = new THREE.MeshPhysicalMaterial({
            color: 0xffffff,
            metalness: 1.0,
            roughness: 0.05,
            envMap: cubeCamera.renderTarget.texture,
        });
        chromeBowl = new THREE.Mesh(bowlGeo, chromeMat);
        chromeBowl.position.y = ROOM_HEIGHT * 0.3;
        scene.add(chromeBowl);

        // Water Surface
        const waterGeo = new THREE.CircleGeometry(ROOM_SIZE / 2,
64);
        const waterMat = new THREE.MeshPhysicalMaterial({
            color: 0x223344,
            transmission: 1.0,
            thickness: 1.5,
```

```
                roughness: 0.1,
            });
            water = new THREE.Mesh(waterGeo, waterMat);
            water.rotation.x = -Math.PI / 2;
            water.position.y = ROOM_HEIGHT * 0.3;
            scene.add(water);

            // Water ripple shader
            water.material.onBeforeCompile = (shader) => {
                 shader.uniforms.u_time = { value: 0.0 };
                 shader.uniforms.u_bhPos1 = { value: new
THREE.Vector3() };
                 shader.uniforms.u_bhPos2 = { value: new
THREE.Vector3() };
                 shader.uniforms.u_mergedBhPos = { value: new
THREE.Vector3() };
                 shader.uniforms.u_waveAmplitude = { value: 0.0 };
                 shader.uniforms.u_phase = { value: 0.0 };

                 shader.vertexShader = 'uniform float u_time;\n' +
                    'uniform vec3 u_bhPos1;\n' +
                    'uniform vec3 u_bhPos2;\n' +
                    'uniform vec3 u_mergedBhPos;\n' +
                    'uniform float u_waveAmplitude;\n' +
                    'uniform float u_phase;\n' +
                    shader.vertexShader;

                 shader.vertexShader = shader.vertexShader.replace(
                    '#include <begin_vertex>',
                    `#include <begin_vertex>

                    float waveStrength = 0.0;
                    float freq = 0.1;
                    vec3 worldPos = (modelMatrix * vec4(position,
1.0)).xyz;

                    if (u_phase < 0.5) {
                        float dist1 = length(worldPos.xz -
u_bhPos1.xz);
                        float dist2 = length(worldPos.xz -
u_bhPos2.xz);
                        waveStrength += sin(dist1 * freq - u_time *
10.0) / (dist1 * 0.1 + 1.0);
                        waveStrength += sin(dist2 * freq - u_time *
10.0) / (dist2 * 0.1 + 1.0);
                    } else {
                        float dist = length(worldPos.xz -
u_mergedBhPos.xz);
```

```
                        waveStrength += sin(dist * freq - u_time *
10.0) / (dist * 0.1 + 1.0);
                    }
                    transformed.y += waveStrength * u_waveAmplitude *
2.0;
                    `
                );
                water.userData.shader = shader;
            };
        }

        function createMergerFlare() {
            const geo = new THREE.BufferGeometry();
            geo.setAttribute('position', new
THREE.Float32BufferAttribute(new Float32Array(MERGER_FLARE_PARTICLES *
3), 3));
            const mat = new THREE.PointsMaterial({ color: 0xfff0b3,
size: 0.3, transparent: true, blending: THREE.AdditiveBlending,
depthWrite: false, opacity: 0 });
            mergerFlare = new THREE.Points(geo, mat);
            mergerFlare.userData.particles = [];
            for(let i=0; i<MERGER_FLARE_PARTICLES; i++) {
                mergerFlare.userData.particles.push({ vel: new
THREE.Vector3(), life: 0 });
            }
            scene.add(mergerFlare);
        }

        function setupPostProcessing() {
            composer = new EffectComposer(renderer);
            composer.addPass(new RenderPass(scene, camera));
            wavePass = new ShaderPass(GravitationalWaveShader());
            composer.addPass(wavePass);
            lensingPass = new ShaderPass(LensingShader());
            composer.addPass(lensingPass);
            bloomPass = new UnrealBloomPass(new
THREE.Vector2(window.innerWidth, window.innerHeight), 1.0, 0.6, 0.1);
            composer.addPass(bloomPass);
        }

        function LensingShader() { return { uniforms: { tDiffuse: {
value: null }, resolution: { value: new THREE.Vector2() }, bhPos1: {
value: new THREE.Vector2(0.5, 0.5) }, bhPos2: { value: new
THREE.Vector2(0.5, 0.5) }, bhRadius1: { value: 0.0 }, bhRadius2: {
value: 0.0 }, mass1: { value: 0.0 }, mass2: { value: 0.0 }, },
vertexShader: `varying vec2 vUv; void main() { vUv = uv; gl_Position =
projectionMatrix * modelViewMatrix * vec4( position, 1.0 ); }`,
fragmentShader: `
```

```
                    uniform sampler2D tDiffuse; uniform vec2
resolution;
                    uniform vec2 bhPos1; uniform vec2 bhPos2;
                    uniform float bhRadius1; uniform float bhRadius2;
                    uniform float mass1; uniform float mass2;
                    varying vec2 vUv;
                    void main() {
                        vec2 aspectUv = vUv * vec2(resolution.x /
resolution.y, 1.0);
                        vec2 aspectBh1 = bhPos1 * vec2(resolution.x /
resolution.y, 1.0);
                        vec2 aspectBh2 = bhPos2 * vec2(resolution.x /
resolution.y, 1.0);
                        vec2 to1 = aspectBh1 - aspectUv; vec2 to2 =
aspectBh2 - aspectUv;
                        float dist1 = length(to1); float dist2 =
length(to2);
                        float pull1 = (mass1 / 10.0) * bhRadius1 *
bhRadius1 / dist1;
                        float pull2 = (mass2 / 10.0) * bhRadius2 *
bhRadius2 / dist2;
                        vec2 offset = normalize(to1) * pull1 +
normalize(to2) * pull2;
                        offset.x /= resolution.x / resolution.y;
                        vec4 color = texture2D(tDiffuse, vUv +
offset);
                        if (dist1 < bhRadius1) { color = vec4(0.0,
0.0, 0.0, 1.0); }
                        if (dist2 < bhRadius2) { color = vec4(0.0,
0.0, 0.0, 1.0); }
                        gl_FragColor = color;
                    }` }; }
        function GravitationalWaveShader() { return { uniforms: {
tDiffuse: { value: null }, time: { value: 0.0 }, origin: { value: new
THREE.Vector2(0.5, 0.5) }, amplitude: { value: 0.0 }, frequency: {
value: 30.0 }, }, vertexShader: `varying vec2 vUv; void main() { vUv =
uv; gl_Position = projectionMatrix * modelViewMatrix * vec4( position,
1.0 ); }`, fragmentShader: `
                    uniform sampler2D tDiffuse; uniform float time;
                    uniform vec2 origin; uniform float amplitude;
uniform float frequency;
                    varying vec2 vUv;
                    void main() {
                        vec2 d = vUv - origin;
                        float dist = length(d);
                        float angle = atan(d.y, d.x);
                        float quadrupole = cos(angle * 2.0);
                        float wave = sin(dist * frequency - time *
```

```
15.0) * amplitude * smoothstep(0.0, 0.8, dist) * quadrupole;
                        vec2 offset = normalize(d) * wave;
                        gl_FragColor = texture2D(tDiffuse, vUv +
offset);
                    }` }; }

        function setupUI() {
            const massRatioSlider =
document.getElementById('massRatioSlider');
            const massRatioValue =
document.getElementById('massRatioValue');
            const separationSlider =
document.getElementById('separationSlider');
            const separationValue =
document.getElementById('separationValue');
            const timeScaleSlider =
document.getElementById('timeScaleSlider');
            const timeScaleValue =
document.getElementById('timeScaleValue');
            const resetBtn = document.getElementById('resetBtn');
            const playPauseBtn =
document.getElementById('playPauseBtn');
            const playIcon = document.getElementById('playIcon');
            const pauseIcon = document.getElementById('pauseIcon');

            massRatioSlider.addEventListener('input', e => {
sim.config.massRatio = parseFloat(e.target.value);
massRatioValue.textContent = sim.config.massRatio.toFixed(2); });
            separationSlider.addEventListener('input', e => {
sim.config.initialSeparation = parseFloat(e.target.value);
separationValue.textContent = sim.config.initialSeparation.toFixed(0);
});
            timeScaleSlider.addEventListener('input', e => {
                const value = parseFloat(e.target.value);
                sim.timeScale = value;
                timeScaleValue.textContent = `${value.toFixed(1)}x`;
            });

            resetBtn.addEventListener('click', resetSimulation);
            playPauseBtn.addEventListener('click', () => {
                sim.physicsPaused = !sim.physicsPaused;
                playIcon.classList.toggle('hidden',
!sim.physicsPaused);
                pauseIcon.classList.toggle('hidden',
sim.physicsPaused);
                playPauseBtn.title = sim.physicsPaused ? 'Play
Physics' : 'Pause Physics';
            });
```

```
        }

        function resetSimulation() {
            if (sim.bh1) { scene.remove(sim.bh1.mesh, sim.bh1.disk); }
            if (sim.bh2) { scene.remove(sim.bh2.mesh, sim.bh2.disk); }
            if (sim.mergedBh) { scene.remove(sim.mergedBh.mesh,
sim.mergedBh.disk); }
            if (mergerFlare) { mergerFlare.visible = false; }

            sim.phase = 'INSPIRAL';
            sim.phaseTime = 0;
            sim.progress = 0;
            sim.wallWaveAmplitude = 0.0;

            const m1 = 12;
            const m2 = m1 * sim.config.massRatio;
            const totalMass = m1 + m2;
            const sep = sim.config.initialSeparation;
            const verticalCenter = ROOM_HEIGHT * 0.4;

            const pos1 = new THREE.Vector3(sep * (m2 / totalMass),
verticalCenter, 0);
            const pos2 = new THREE.Vector3(-sep * (m1 / totalMass),
verticalCenter, 0);

            const orbitalVel = Math.sqrt(G * totalMass / sep);
            const vel1 = new THREE.Vector3(0, 0, orbitalVel * (m2 /
totalMass));
            const vel2 = new THREE.Vector3(0, 0, -orbitalVel * (m1 /
totalMass));

            sim.bh1 = new BlackHole(m1, pos1, vel1, 0xffcc33,
0xff8800);
            sim.bh2 = new BlackHole(m2, pos2, vel2, 0x66aaff,
0x0055ff);
            sim.mergedBh = new BlackHole(1, new THREE.Vector3(), new
THREE.Vector3(), 0xffaaff, 0xcc88ff);
            sim.mergedBh.setVisible(false);

            initDisk(sim.bh1);
            initDisk(sim.bh2);

            wavePass.uniforms.amplitude.value = 0;
            updateStatusUI();
        }

        function initDisk(bh) {
            const particles = bh.disk.userData.particles;
```

```
            const positions =
bh.disk.geometry.attributes.position.array;
            const colors = bh.disk.geometry.attributes.color.array;
            const color1 = new
THREE.Color(bh.mesh.children[0].children[0].material.uniforms.glowColo
r.value);
            const color2 = color1.clone().lerp(new
THREE.Color(0x000000), 0.5);

            for(let i=0; i<particles.length; i++) {
                const p = particles[i];
                const r = bh.radius * 2.5 + Math.random() * bh.radius
* 8;
                p.baseRadius = r;
                p.angle = Math.random() * Math.PI * 2;
                p.life = 1;

                const i3 = i * 3;
                positions[i3] = Math.cos(p.angle) * r;
                positions[i3+1] = (Math.random() - 0.5) * bh.radius *
0.2 * (1 - (r - bh.radius*2.5)/(bh.radius*8));
                positions[i3+2] = Math.sin(p.angle) * r;

                const mix = Math.pow((r -
bh.radius*2.5)/(bh.radius*8), 0.5);
                const c = color1.clone().lerp(color2, mix);
                colors[i3] = c.r; colors[i3+1] = c.g; colors[i3+2] =
c.b;
            }
            bh.disk.geometry.attributes.position.needsUpdate = true;
            bh.disk.geometry.attributes.color.needsUpdate = true;
        }

        function updatePhysics(dt) {
            if (sim.phase !== 'INSPIRAL') return;
            const dPos = new THREE.Vector3().subVectors(sim.bh2.pos,
sim.bh1.pos);
            const rSq = dPos.lengthSq();
            const r = Math.sqrt(rSq);
            const mergerDist = (sim.bh1.radius + sim.bh2.radius);
            if (r < mergerDist) { triggerMerger(); return; }
            const forceMag = G * sim.bh1.mass * sim.bh2.mass / rSq;
            const forceVec =
dPos.normalize().multiplyScalar(forceMag);
            sim.bh1.vel.add(forceVec.clone().multiplyScalar(dt /
sim.bh1.mass));
            sim.bh2.vel.add(forceVec.clone().multiplyScalar(-dt /
sim.bh2.mass));
```

```
            const energyLossFactor = (32/5) * (Math.pow(G, 4) /
Math.pow(C, 5)) * (sim.bh1.mass * sim.bh2.mass * (sim.bh1.mass +
sim.bh2.mass)) / Math.pow(r, 4);
            sim.bh1.vel.multiplyScalar(1.0 - energyLossFactor * dt *
0.001);
            sim.bh2.vel.multiplyScalar(1.0 - energyLossFactor * dt *
0.001);
            sim.bh1.update(dt);
            sim.bh2.update(dt);
            sim.progress = 1.0 - (r - mergerDist) /
(sim.config.initialSeparation - mergerDist);
            sim.wallWaveAmplitude = 0.01 + sim.progress * 0.5;
        }

        function triggerMerger() {
            sim.phase = 'MERGER';
            sim.phaseTime = 0;
            const p1 =
sim.bh1.vel.clone().multiplyScalar(sim.bh1.mass);
            const p2 =
sim.bh2.vel.clone().multiplyScalar(sim.bh2.mass);
            const finalMomentum = p1.add(p2);
            const com1 =
sim.bh1.pos.clone().multiplyScalar(sim.bh1.mass);
            const com2 =
sim.bh2.pos.clone().multiplyScalar(sim.bh2.mass);
            const finalCoM = com1.add(com2);
            const initialTotalMass = sim.bh1.mass + sim.bh2.mass;
            const massLoss = 0.05;
            sim.mergedBh.mass = initialTotalMass * (1 - massLoss);
            sim.mergedBh.radius = Math.pow(sim.mergedBh.mass, 0.7) *
0.2;
            finalCoM.divideScalar(initialTotalMass);
            finalMomentum.divideScalar(sim.mergedBh.mass);
            sim.mergedBh.pos.copy(finalCoM);
            sim.mergedBh.vel.copy(finalMomentum);
            sim.bh1.setVisible(false);
            sim.bh2.setVisible(false);
            sim.mergedBh.setVisible(true);
            sim.mergedBh.mesh.scale.setScalar(0.1);
            initDisk(sim.mergedBh);
            bloomPass.strength = 1.5;
            wavePass.uniforms.amplitude.value = 0.05;
            wavePass.uniforms.frequency.value = 60.0;
            sim.wallWaveAmplitude = 1.0;
            triggerFlare(finalCoM);
            sim.progress = 0.9;
            updateStatusUI();
```

```
        }

        function triggerFlare(position) {
            mergerFlare.position.copy(position);
            mergerFlare.material.opacity = 1.0;
            mergerFlare.visible = true;
            const particles = mergerFlare.userData.particles;
            const positions =
mergerFlare.geometry.attributes.position.array;
            for(let i=0; i<MERGER_FLARE_PARTICLES; i++) {
                const p = particles[i];
                p.vel.set(Math.random()-0.5, Math.random()-0.5,
Math.random()-0.5).normalize().multiplyScalar(Math.random() * 20);
                p.life = 1.0;
                positions[i*3] = positions[i*3+1] = positions[i*3+2] =
0;
            }
        }

        function updateMerger(dt) {
            sim.phaseTime += dt;
            const scale = THREE.MathUtils.lerp(0.1,
sim.mergedBh.radius, Math.min(1, sim.phaseTime * 2));
            sim.mergedBh.mesh.scale.setScalar(scale);
            if (mergerFlare.material.opacity > 0) {
                mergerFlare.material.opacity -= dt * 0.8;
                const positions =
mergerFlare.geometry.attributes.position.array;
                const particles = mergerFlare.userData.particles;
                for(let i=0; i<particles.length; i++) {
                    const p = particles[i];
                    positions[i*3] += p.vel.x * dt;
                    positions[i*3+1] += p.vel.y * dt;
                    positions[i*3+2] += p.vel.z * dt;
                }
                mergerFlare.geometry.attributes.position.needsUpdate =
true;
            } else {
                mergerFlare.visible = false;
            }
            if (sim.phaseTime > 1.5) {
                sim.phase = 'RINGDOWN';
                sim.phaseTime = 0;
                sim.progress = 0.95;
                updateStatusUI();
            }
        }
```

```
        function updateRingdown(dt) {
            sim.phaseTime += dt;
            if (bloomPass.strength > 1.0) bloomPass.strength -= 1.5 *
dt;
            if (wavePass.uniforms.amplitude.value > 0) {
                wavePass.uniforms.amplitude.value *= (1 - 1.5 * dt);
                wavePass.uniforms.frequency.value *= (1 - 1.0 * dt);
            }
            if (sim.wallWaveAmplitude > 0) {
                sim.wallWaveAmplitude *= (1 - 1.2 * dt);
            }
            sim.mergedBh.update(dt);
            sim.progress = Math.min(1.0, 0.95 + sim.phaseTime * 0.05);
        }

        function updateDisk(bh, dt, otherBh = null) {
            const particles = bh.disk.userData.particles;
            const positions =
bh.disk.geometry.attributes.position.array;
            for(let i=0; i<particles.length; i++) {
                const p = particles[i];
                if (p.life <= 0) continue;
                const i3 = i * 3;
                const currentPos = new THREE.Vector3(positions[i3],
positions[i3+1], positions[i3+2]);
                const r = currentPos.length();
                const orbitalSpeed = Math.sqrt(G * bh.mass /
Math.max(0.1, r));
                const tangent = new THREE.Vector3(-currentPos.z, 0,
currentPos.x).normalize();
                p.vel = tangent.multiplyScalar(orbitalSpeed);
                const pull =
currentPos.clone().normalize().multiplyScalar(-G * bh.mass /
Math.max(0.1, r*r));
                p.vel.add(pull.multiplyScalar(dt));
                if(otherBh) {
                    const toOther = new
THREE.Vector3().subVectors(otherBh.pos,
bh.pos.clone().add(currentPos));
                    const tidalForce = G * otherBh.mass /
toOther.lengthSq();

p.vel.add(toOther.normalize().multiplyScalar(tidalForce * dt));
                }
                positions[i3] += p.vel.x * dt;
                positions[i3+1] += p.vel.y * dt;
                positions[i3+2] += p.vel.z * dt;
                if (r < bh.radius * 1.5) { p.life = 0; positions[i3] =
```

```
1e6; }
            }
            bh.disk.geometry.attributes.position.needsUpdate = true;
        }

        function updateLensing() {
            const project = (bh, targetPos) => {
                if (!bh || !bh.mesh.visible) return 0;
                const p = bh.pos.clone();
                const vector = p.project(camera);
                targetPos.set((vector.x + 1) / 2, (vector.y + 1) / 2);
                const edgePointWorld = bh.mesh.localToWorld(new
THREE.Vector3(1, 0, 0));
                const edgePointNDC = edgePointWorld.project(camera);
                return new THREE.Vector2(edgePointNDC.x,
edgePointNDC.y).distanceTo(new THREE.Vector2(vector.x, vector.y));
            }

lensingPass.uniforms.resolution.value.set(window.innerWidth *
window.devicePixelRatio, window.innerHeight *
window.devicePixelRatio);
            if (sim.phase === 'INSPIRAL') {
                lensingPass.uniforms.bhRadius1.value =
project(sim.bh1, lensingPass.uniforms.bhPos1.value);
                lensingPass.uniforms.bhRadius2.value =
project(sim.bh2, lensingPass.uniforms.bhPos2.value);
                lensingPass.uniforms.mass1.value = sim.bh1.mass;
                lensingPass.uniforms.mass2.value = sim.bh2.mass;
            } else {
                lensingPass.uniforms.bhRadius1.value =
project(sim.mergedBh, lensingPass.uniforms.bhPos1.value);
                lensingPass.uniforms.bhRadius2.value = 0;
                lensingPass.uniforms.mass1.value = sim.mergedBh.mass;
                lensingPass.uniforms.mass2.value = 0;
            }
        }

        function updateStatusUI() {
            const textEl = document.getElementById('statusText');
            const progressEl = document.getElementById('progressBar');
            if (sim.phase === 'INSPIRAL') textEl.textContent =
'Inspiral';
            else if (sim.phase === 'MERGER') textEl.textContent =
'MERGER!';
            else if (sim.phase === 'RINGDOWN') textEl.textContent =
'Ringdown';
            progressEl.style.width = `${sim.progress * 100}%`;
        }
```

```
        // --- Animation Loop ---
        function animate() {
            requestAnimationFrame(animate);
            const dt = Math.min(0.016, clock.getDelta());
            const elapsedTime = clock.getElapsedTime();

            controls.update();

            if (!sim.physicsPaused && dt > 0) {
                const simDt = dt * sim.timeScale;
                if (sim.phase === 'INSPIRAL') { updatePhysics(simDt);
updateDisk(sim.bh1, simDt, sim.bh2); updateDisk(sim.bh2, simDt,
sim.bh1); }
                else if (sim.phase === 'MERGER') {
updateMerger(simDt); sim.mergedBh.update(simDt);
updateDisk(sim.mergedBh, simDt); }
                else if (sim.phase === 'RINGDOWN') {
updateRingdown(simDt); sim.mergedBh.update(simDt); }
            }

            coloredLights.forEach((light, i) => {
                const angle = elapsedTime * 0.3 + (i /
coloredLights.length) * Math.PI * 2;
                light.position.x = Math.cos(angle) * ROOM_SIZE * 0.4;
                light.position.z = Math.sin(angle) * ROOM_SIZE * 0.4;
            });

            const viewVector = camera.position.clone().normalize();
            if (sim.bh1 && sim.bh1.mesh.visible)
sim.bh1.mesh.children[0].children[0].material.uniforms.viewVector.valu
e = viewVector;
            if (sim.bh2 && sim.bh2.mesh.visible)
sim.bh2.mesh.children[0].children[0].material.uniforms.viewVector.valu
e = viewVector;
            if (sim.mergedBh && sim.mergedBh.mesh.visible)
sim.mergedBh.mesh.children[0].children[0].material.uniforms.viewVector
.value = viewVector;

            updateLensing();
            wavePass.uniforms.time.value += dt * sim.timeScale;
            if(sim.phase === 'INSPIRAL' && sim.bh1 && sim.bh2) {
                const com =
sim.bh1.pos.clone().multiplyScalar(sim.bh1.mass).add(sim.bh2.pos.clone
().multiplyScalar(sim.bh2.mass)).divideScalar(sim.bh1.mass +
sim.bh2.mass);
                const screenPos = com.project(camera);
                wavePass.uniforms.origin.value.set((screenPos.x+1)/2,
```

```
(screenPos.y+1)/2);
            } else if (sim.mergedBh && sim.mergedBh.pos) {
                const screenPos =
sim.mergedBh.pos.clone().project(camera);
                wavePass.uniforms.origin.value.set((screenPos.x+1)/2,
(screenPos.y+1)/2);
            }

            if (water && water.userData.shader) {
                chromeBowl.visible = false;
                water.visible = false;
                cubeCamera.update(renderer, scene);
                chromeBowl.visible = true;
                water.visible = true;

                const uniforms = water.userData.shader.uniforms;
                uniforms.u_time.value = elapsedTime;
                uniforms.u_waveAmplitude.value =
sim.wallWaveAmplitude;

                if (sim.phase === 'INSPIRAL') {
                    uniforms.u_phase.value = 0.0;
                    if (sim.bh1)
uniforms.u_bhPos1.value.copy(sim.bh1.pos);
                    if (sim.bh2)
uniforms.u_bhPos2.value.copy(sim.bh2.pos);
                } else {
                    uniforms.u_phase.value = 1.0;
                    if (sim.mergedBh)
uniforms.u_mergedBhPos.value.copy(sim.mergedBh.pos);
                }
            }

            updateStatusUI();
            composer.render();
        }

        window.addEventListener('resize', () => {
            camera.aspect = window.innerWidth / window.innerHeight;
            camera.updateProjectionMatrix();
            renderer.setSize(window.innerWidth, window.innerHeight);
            composer.setSize(window.innerWidth, window.innerHeight);
        });
    </script>
</body>
</html>
```