Protocol for Generative WebGL Simulation Architecture

Version: 2.0 **Context:** Three.js / Physics-Based Animation / Generative Code **Target Agent:** Large Language Models (LLMs) generating single-file HTML simulations.

1.0 Executive Summary

This protocol defines the architectural standard for generating high-fidelity, physics-based simulations within a single executable file. The methodology enforces a **"System of Specialists"** approach, compartmentalizing code into distinct functional domains (Environment, Assets, Physics, Logic) to mitigate Context Drift and State Entanglement.

Historical iteration analysis (e.g., *Project Eagle*) indicates that monolithic code generation results in geometric primitives, floating artifacts, and coordinate system failures. Adherence to this modular standard is mandatory.

2.0 Phase I: The World Anchor (Environment & Coordinates)

Objective: Elimination of spatial disorientation and lighting artifacts.

A common failure mode in generative simulation is the "Void State," where objects are rendered without spatial reference, leading to scale and orientation errors. The simulation environment must be initialized as a verified stage prior to the introduction of active agents.

2.1 Initialization Standards

The following configuration is required for all simulations: 1. **Coordinate System:** Standard Y-Up configuration. The origin `(0,0,0)` is defined as the ground plane center. 2. **Scale Authority:** Explicitly define `1.0 unit = 1.0 meter`. 3. **Lighting Rig:** A dual-source setup is mandatory to prevent flat rendering. * `HemisphereLight`: Ambient illumination (Sky/Ground color). * `DirectionalLight`: Key light with shadow casting enabled. 4. **Reference Geometry:** A `GridHelper` must be included during the initial render to validate scale.

2.2 Code Implementation: Standard Anchor

```
/**
 * @brief Initializes the simulation environment anchor.
 * @param scene (THREE.Scene): The active scene graph.
```

```
 * @return void
 */
function initWorldAnchor(scene) {
    // 1. Ground Plane (Reference Surface)
    // Optimization: Use basic geometry for the floor to save draw calls
    const groundGeo = new THREE.CircleGeometry(800, 64);
    const groundMat = new THREE.MeshStandardMaterial({
        color: 0x3a5f0b, // Biological/Natural context
        roughness: 1.0
    });
    const ground = new THREE.Mesh(groundGeo, groundMat);
    ground.rotation.x = -Math.PI / 2; // Rotate to X-Z plane
    ground.receiveShadow = true;
    scene.add(ground);

    // 2. Lighting Rig
    const hemiLight = new THREE.HemisphereLight(0xffffff, 0x444444, 0.6);
    scene.add(hemiLight);

    const dirLight = new THREE.DirectionalLight(0xffffff, 1.2);
    dirLight.position.set(50, 100, 50); // High angle sun
    dirLight.castShadow = true;
    scene.add(dirLight);
}
```

3.0 Phase II: The Asset Factory (Compound Geometry)

Objective: Prevention of "Primitive Collapse" (low-fidelity cubes/spheres).

LLMs default to minimum viable geometry (e.g., representing a bird as a single cone). Assets must be constructed as hierarchical `THREE.Group` objects composed of multiple sub-primitives. This is the **Compound Primitive Rule**.

3.1 Construction Rules

1. **Hierarchy:** The root object is a `Group`. All mesh components are children of this group.
2. **Materiality:** Use `MeshStandardMaterial` to leverage the lighting rig. Define `roughness` and `metalness` appropriate to the material (e.g., Feathers vs. Concrete).
3. **Pivot Points:** Moving parts (wings, wheels, rotors) must be nested in a parent Group positioned at the mechanical pivot point, not the geometric center.

3.2 Code Implementation: Hierarchical Actor

```
/**
 * @brief Represents a compound biological actor (Eagle).
 * @details Constructs a bird using hierarchical grouping for articulation.
 */
class AvianActor {
    constructor(scene) {
        this.mesh = new THREE.Group();

        // Material Library
        const matFeather = new THREE.MeshStandardMaterial({
            color: 0x4a3728, // Dark Brown
            roughness: 0.8
        });

        // 1. Torso (Root body)
        const body = new THREE.Mesh(new THREE.CapsuleGeometry(0.4, 1.0, 4, 8), matFeather);
        body.rotation.x = Math.PI / 2; // Align with forward vector
        this.mesh.add(body);

        // 2. Articulated Wing (Left)
        // Optimization: Create a pivot group to handle rotation at the shoulder
        this.leftWing = new THREE.Group();
        this.leftWing.position.set(0.3, 0.1, 0.2); // Shoulder position

        const wingGeo = new THREE.BoxGeometry(2.0, 0.05, 0.8);
        wingGeo.translate(1.0, 0, 0); // Offset geometry so pivot is at the edge

        const wingMesh = new THREE.Mesh(wingGeo, matFeather);
        this.leftWing.add(wingMesh);
        this.mesh.add(this.leftWing);

        scene.add(this.mesh);
    }

    /**
     * @brief Updates internal animation state.
     * @param dt (float): Delta time in seconds.
     */
    updateAnimation(dt) {
        // Sine wave oscillation for organic movement
        const angle = Math.sin(Date.now() * 0.01) * 0.5;
        this.leftWing.rotation.z = -angle;
    }
}
```

4.0 Phase III: The Kinematic Kernel (Physics Engine)

Objective: Decoupling of visual rendering from physical calculation.

A primary failure mode in generated code is "Animation-Physics Entanglement," where movement is hardcoded to frame updates rather than time deltas. This results in inconsistent speeds and floating debris.

4.1 Physics Principles

1. **Delta Time (`dt`):** All motion must be scaled by `dt`.
2. **Explicit Gravity:** Objects do not fall unless a gravity constant acts upon their Y-velocity.
   - *Heavy Objects (Concrete):* High Gravity (~30-60 m/s^2 visual scale), Low Drag.
   - *Light Objects (Feathers):* Low Gravity (~2-5 m/s^2), High Drag.
3. **State Separation:** Physics properties (`velocity`, `mass`) are stored in `userData` or a dedicated physics class, separate from the `Mesh` properties.

4.2 Code Implementation: Particle Physics

```
/**
 * @brief Updates the physics state of light particles (Feathers/Dust).
 * @param particle The particle object containing physics state.
 * @param dt Delta time in seconds.
 */
function updateParticlePhysics(particle, dt) {
    if(!particle.visible) return;

    // 1. Integration (Position += Velocity * dt)
    particle.mesh.position.addScaledVector(particle.vel, dt);
    particle.life -= dt;

    if (particle.type === 'FEATHER') {
        // 2. Aerodynamics
        particle.vel.y -= 2.0 * dt;        // Low Gravity (Floaty)
        particle.vel.multiplyScalar(0.95); // High Drag (Air resistance)

        // 3. Chaotic Rotation (Flutter)
        particle.mesh.rotation.x += particle.rotVel.x * dt;
        particle.mesh.rotation.y += particle.rotVel.y * dt;
    }
    else if (particle.type === 'STONE') {
        // Ballistic Trajectory
        particle.vel.y -= 9.8 * dt; // Standard Gravity
```

```
    }

    // 4. Boundary Condition (Ground Plane)
    if(particle.mesh.position.y < 0.1) {
        particle.mesh.position.y = 0.1;
        particle.vel.set(0,0,0); // Friction stop
    }
}
```

5.0 Phase IV: The Simulation Director (State Machine)

Objective: Synchronization of disparate events (Impact, Reaction, FX).

Complex simulations must be governed by a central State Machine. Logic branching (e.g., "If distance < 1.0") must occur within the Director, not within the individual Actor classes.

5.1 State Definitions

- **LOITER/IDLE:** The system holds a steady state (e.g., circling pattern).
- **KINEMATIC PHASE:** Active movement toward a goal (e.g., Dive/Attack).
- **CONTACT EVENT:** Discrete logic tick where collision is detected. Triggers FX spawning and state transition.
- **TERMINAL PHASE:** Post-event simulation (e.g., Feeding, Collapse).

5.2 Code Implementation: Logic Loop

```
/**
 * @brief Main simulation loop handling state transitions.
 * @param simState Object containing current pattern and flags.
 * @param actors Dictionary of active scene actors.
 */
function renderLoop(simState, actors, clock) {
    const dt = clock.getDelta();
    const now = clock.getElapsedTime();

    // State: Loiter
    if (simState.pattern === 'LOITER') {
        updateLoiterPath(actors.predator, now);
    }
    // State: Attack Run
    else if (simState.pattern === 'ATTACK' && !simState.contact) {
        const dist = actors.predator.mesh.position.distanceTo(actors.prey.position);
```

```
        // Trigger: Contact Event
        if (dist < 1.5) {
            simState.contact = true;

            // 1. Reaction
            actors.prey.triggerCaptureState();

            // 2. FX Generation
            actors.fx.spawnFeathers(actors.prey.position, 20);

            // 3. Pose Update
            actors.predator.setLandedPose();
        } else {
            // Continuation
            updateDivePath(actors.predator, actors.prey.position, now);
        }
    }

    // Continuous Physics Update (Run regardless of state)
    actors.fx.update(dt);
    actors.predator.updateAnimation(dt);
}
```

6.0 Iteration Mitigation Checklist (Failure Analysis)

Before generating code, the following failure modes must be explicitly ruled out.

Failure Mode	Symptom	Engineering Solution
Coordinate Drift	Object components separate during movement.	**Rule:** Apply movement vectors to the parent `Group` only. Never move child meshes independently unless animating local articulation.
Floating Debris	Particles hang in the air after an explosion.	**Rule:** Ensure the `update(dt)` loop applies gravity (`vel.y -= g * dt`) every frame. Remove air drag for heavy objects.

Failure Mode	Symptom	Engineering Solution
Offset Geometry	Fire/FX appears distant from the target object.	**Rule:** Particle spawn logic must calculate `Target.WorldPosition + LocalOffset` to align coordinate spaces.
Animation Speed	Collapse/Motion feels like "toy scale."	**Rule:** Physics constants must be tuned for scale. Large objects require slower acceleration curves (Square-Cube Law visual perception).
Primitive Geometry	Objects look like red boxes or grey spheres.	**Rule:** Enforce the "Compound Primitive" rule. A minimum of 3 geometric primitives is required to define any Actor.

7.0 Safety & Optimization Footer

Failure Analysis: * **Performance:** High particle counts (>1000) without instancing may cause frame drops on mobile devices. Limit particle pools. * **Memory:** Geometries and Materials should be created once and cloned or shared, not recreated every frame. * **Context:** This protocol assumes a standard browser environment with WebGL support.

Governing Assumptions: * Simulations approximate Newtonian physics but prioritize visual fidelity over mathematical precision. * Collision detection is distance-based (spherical approximation), not mesh-based.